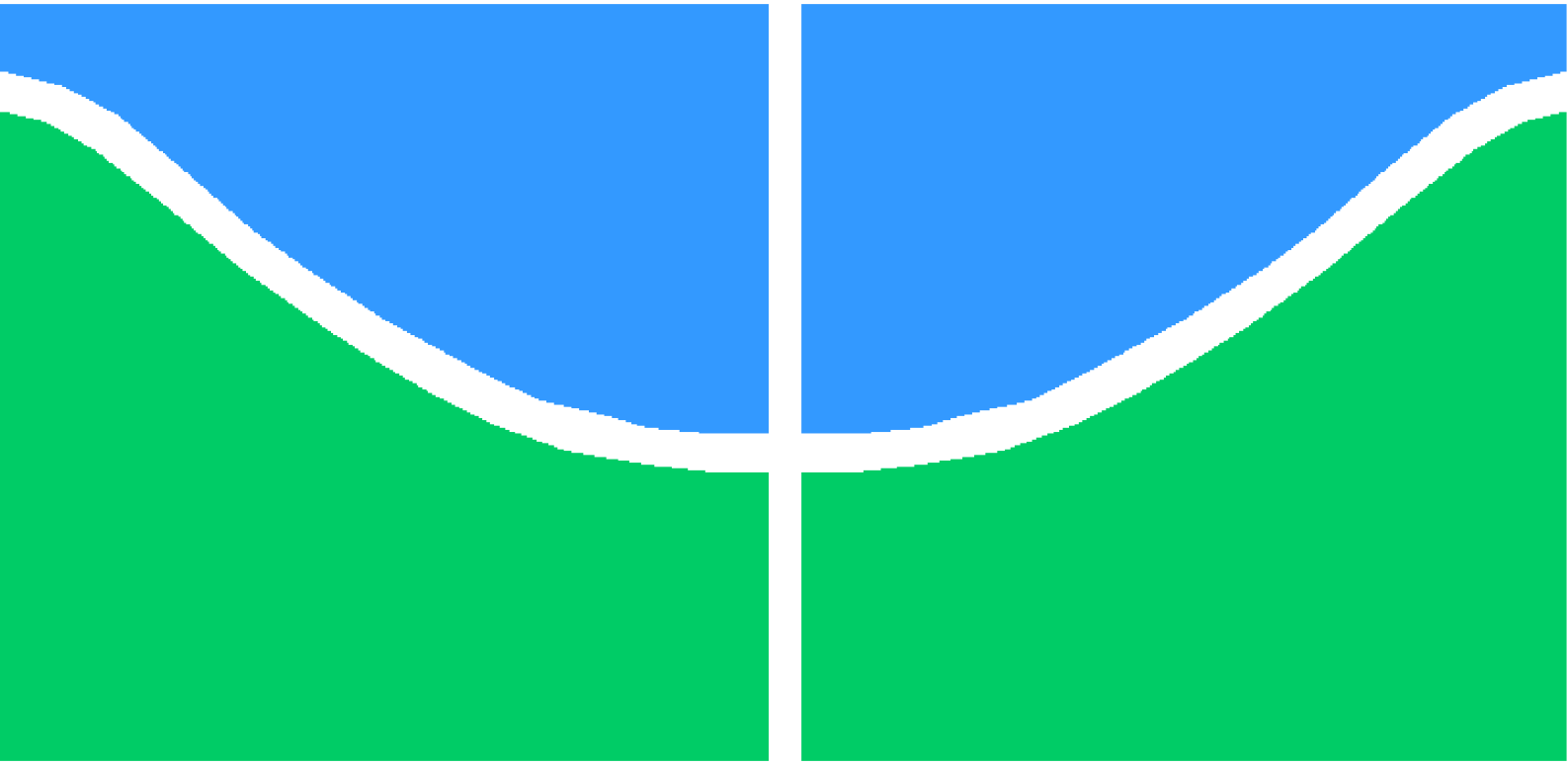

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

Análise estática de vulnerabilidades em software sob múltiplas abordagens

Autor: Alexandre Almeida Barbosa
Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Brasília, DF
2015

Alexandre Almeida Barbosa

Análise estática de vulnerabilidades em software sob múltiplas abordagens

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Brasília, DF

2015

Alexandre Almeida Barbosa
Análise estática de vulnerabilidades em software sob múltiplas abordagens/ Alexandre Almeida Barbosa. – Brasília, DF, 2015-
100 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2015.

1. Engenharia de Software. 2. Análise de Fraquezas de código. I. Prof. Dr. Luiz Augusto Fontes Laranjeira. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Análise estática de vulnerabilidades em software sob múltiplas abordagens

CDU 02:141:005.6

Alexandre Almeida Barbosa

Análise estática de vulnerabilidades em software sob múltiplas abordagens

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Trabalho aprovado. Brasília, DF, 13 de março de 2015:

Prof. Dr. Luiz Augusto Fontes Laranjeira
Orientador

Prof. Dr. Paulo Roberto Miranda Meirelles
Convidado 1

Prof. Me. Hilmer Rodrigues Neri
Convidado 2

Brasília, DF
2015

Este trabalho é dedicado à minha família,
que me apoia em todos os momentos.
E aos meus amigos,
que souberam se tornar meus irmãos.

Agradecimentos

Agradeço à Universidade de Brasília por me permitir crescer pessoal e profissionalmente, por me oferecer auxílios e suportes que foram essenciais para a minha permanência no Distrito Federal durante a minha graduação, e por me mostrar que, apesar das dificuldades, com as quais também cresci, uma Universidade Pública é um dos maiores ferramentas para o crescimento do país.

Aos professores e alunos que compuseram e compõem o LART, laboratório que me permitiu enxergar a Engenharia de Software como uma ferramenta de transformação humana e que me mostrou que a troca de experiências entre áreas de conhecimento distintas permite alcançar resultados com muito mais relevância para o bem comum.

Agradeço aos professores e alunos que compuseram e compõem o LAPPIS, laboratório que me proporcionou um aprendizado acadêmico que excede em muito as aulas, palestras e trabalhos que tive ao longo de minha graduação e do qual sempre terei orgulho de ter sido integrante. A estes, devo muito dos princípios e éticas profissionais que pretendo levar ao longo da minha vida.

Agradeço à minha mãe, Judite, pela confiança, pelo apoio e pelo amor que tem me oferecido ao longo de toda a minha vida, dizendo sempre que a minha felicidade é o melhor presente que posso oferecer em troca de tudo o que faz por mim. A ela devo a minha vida, o meu caráter e os valores que definem quem me tornei.

Ao meu pai, Luiz, pela confiança, pelo amor e por todas as lições de vida que me ensinou e me ensina sempre, dedicando muito da própria vida, para que eu tenha uma vida melhor e que também seja uma pessoa melhor. A ele devo a minha vida, o meu caráter e os princípios humanos que me definem.

Agradeço aos meus familiares e amigos que, perto ou longe, contribuem de forma única para o meu crescimento enquanto pessoa e sem os quais eu também não seria nada nessa vida.

Por fim, agradeço a Deus, pela família com a qual me presenteou, pelos amigos que me permitiu conhecer e conviver e pelos mestres, acadêmicos ou não, com os quais pude aprender sobre a minha profissão sobre a vida. A Ele devo tudo, simplesmente.

"Ao ser humano cabem os projetos,
mas a resposta pertence ao SENHOR,
Aos olhos humanos são limpos todos os caminhos,
mas é o SENHOR quem avalia os espíritos.
Revela ao SENHOR tuas tarefas,
e teus projetos se realizarão.
(Bíblia Sagrada, Provérbios 16, 1-3)

Resumo

O uso de softwares torna-se, nos dias atuais, cada vez mais presente e necessário. Sejam em instituições privadas, governamentais ou não governamentais, a presença de sistemas que gerenciem dados, transações, produtos e processos, torna crítico o modo como a segurança desses sistemas é tratada. A presença de vulnerabilidades, por sua vez, podem permitir a ocorrência de falhas de segurança, comportamentos inesperados ou mesmo falhas totais de grande impacto financeiro, intelectual e até relacionados à vida humana. Uma abordagem sistemática e eficiente na correção e tratamento de vulnerabilidades possibilita que boa parte dessas sejam evitadas, solucionadas ou mitigadas. O presente trabalho analisa o comportamento de ferramentas que implementam diversos métodos de análise estática de vulnerabilidades. Restringe-se este trabalho à análise de vulnerabilidades relacionadas a *buffer overflow*. Para isso, é feito um levantamento das principais características que definem um *buffer overflow*, das técnicas de análise estática mais conhecidas para detecção dessas vulnerabilidades e de ferramentas que as implementam. Objetiva-se, com isso, analisar o potencial de cada ferramenta para cada característica de um *buffer overflow* analisada.

Palavras-chaves: análise estática. análise híbrida *buffer overflow*. fraquezas de software.

Abstract

The use of software has come nowadays increasingly present and necessary. Whether in private, governmental or non-governmental institutions, the presence of systems that manage data, transactions, products and processes makes it critical how the security of these systems is treated. The presence of vulnerabilities can give room to the occurrence of system security breaches, unexpected behavior or even total system failure leading to financial or intellectual losses and even impacting human life. A systematic and efficient approach in the treatment and correction of vulnerabilities allows most of these to be avoided, remedied or mitigated. This paper analyzes the behavior of tools that implement various methods of vulnerability static analysis. This work is restricted to the analysis of vulnerabilities related to buffer overflow. For this, we made a survey of the main characteristics that define a buffer overflow, of the static analysis techniques widely known to detect these vulnerabilities of the tools that implement them. The purpose is, therefore, evaluate the potential of each tool concerning each buffer overflow characteristic analyzed.

Key-words: static analysis. buffer overflow. software weakness.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AST	*Abstract Sintax Tree* (Árvore de Sintaxe Abstrata)
BSS	*Block Started by Symbol*
C.I.A.	*Confidentiality, Integrity and Availability*
CVE	*Common Vulnerabilities and Exposures*
CWE	*Common Weakness Enumeration*
NIST	*National Institute of Standards and Technology*
SDLC	*Systems Development Life Cycle* (Ciclo de Vida de Desenvolvimento de Sistemas)
TP	*True Positive* (Verdadeiro Positivo)
FP	*False Positive* (Falso Positivo)
FN	*False Negative* (Falso Negativos)

Sumário

Parte I

Introdução

1 Introdução

1.1 Contextualização

Atualmente, softwares sustentam toda a infraestrutura tecnológica que empresas, governos e instituições utilizam para grande parte de suas operações. A prevenção e combate a potenciais ataques maliciosos, ou mesmo falhas por uso ocorridas de modo não intencional, são motivos de preocupação constante. Diversas são as medidas de segurança adotadas, ao longo do ciclo de vida de desses softwares. Todas com o intuito de combater o problema existente(PAUL, 2008). Nesse contexto, a análise de vulnerabilidades insere-se como uma alternativa adicional na busca por softwares mais robustos no que se refere a aspectos de segurança destes. Entretanto, o problema que esse tipo de análise propõe combater mostra-se, na maioria das vezes, muito complexo para a existência de uma solução realmente eficiente. A maioria das ferramentas existentes não consegue identificar, de modo eficaz muitos dos tipos de vulnerabilidades de forma simultânea. Isso ocorre porque essas vulnerabilidades possuem características e comportamentos distintos, não sendo possível identificar, satisfatoriamente, um grande conjunto de vulnerabilidades utilizando apenas uma perspectiva.

1.2 Problema

Identificar as características de detecção vulnerabilidades de software de ferramentas de análise estática de código de código, a partir das caracterísiticas que essas vulnerabilidades possuem.

1.3 Propósito do Estudo

O estudo e desenvolvimento de ferramentas para análise de código está em crescimento constante sendo possível encontrar, atualmente, ferramentas muito eficientes em termos de precisão e acurácia, para a detecção de certas vulnerabilidades, se comparadas a ferramentas mais antigas. As primeiras ferramentas de análise estática, por exemplo, surgiram no final dos anos 70. Estas ainda eram muito falhas no que se refere à não detecção de certas falhas (falso negativo) enquanto, muitas vezes, reportavam falhas onde não haviam (falso positivo)(SHRESTHA, 2013). Além do aprimoramento das técnicas e abordagens utilizadas para detecção de vulnerabilidades, novas perspectivas, e junção de perspectivas estão sendo estudadas e implementadas com essa mesma finalidade.

O presente trabalho tem como propósito analisar o estado da arte da análise estática de softwares e, a partir desse estudo, identificar características distintas de métodos e técnicas de análise, identificando o uso de cada técnica em contextos conhecidos, bem como facilitando a junção de uma ou mais técnicas, de acordo com os principais objetivos da análise feita. Shrestha (2013) afirma que algumas ferramentas podem ser muito eficientes em determinadas categorias de vulnerabilidades, mas são pouco eficientes em outras. A identificação desses pontos fortes e fracos dos principais tipos de perspectiva aqui abordados pode propiciar uma abordagem mais eficiente na detecção dessas vulnerabilidades.

1.4 Objetivo

Analisar os fatores que caracterizam uma dada vulnerabilidade e, utilizando essas características como base, analisar o comportamento de múltiplas técnicas de análise estática, através de ferramentas que as implementam.

1.5 Questões que orientam pesquisa

Com o objetivo de auxiliar e orientar o desenvolvimento do presente trabalho, foram elaboradas as questões de apoio a seguir:

- Quais os aspectos que caracterizam e identificam vulnerabilidades relacionadas a *buffer overflow*?
- Quais os principais aspectos que caracterizam cada abordagem de análise estática apresentada?
- Quais aspectos são redundantes e quais se somam, numa integração entre as abordagens apresentadas?
- Que características são evidenciadas pelas métricas calculadas para cada tipo de abordagem?

1.6 Escopo do Trabalho

O presente trabalho caracteriza-se pela análise do comportamento de ferramentas que implementem diversas metodologias de análise estática de vulnerabilidades. Deve ser utilizada uma metodologia de avaliação que permita a uma análise comparativa imparcial das ferramentas analisadas. tratadas pelo programa. O escopo desse trabalho segue as seguintes premissas:

- Além das abordagens de análise estática, é possível analisar dinamicamente um determinado código. Essa execução analisa o comportamento de cada código analisado, dificultando a replicação dos valores de eficiência em outros contextos. O presente trabalho restringe-se a avaliação das funcionalidades de ferramentas voltadas para a análise estática.

- Existem muitos grupos em que podem ser classificadas as vulnerabilidades existentes em um código. Devido à alta distinção entre as características desses grupos, torna-se inviável analisá-las sob uma mesma perspectiva esperando-se obter resultados semelhantes. Restringe-se, então, à análise de vulnerabilidades relacionadas a *Buffer Overflow*. As razões que justificam a escolha desse grupo de vulnerabilidades estão listadas em CWEs Diretamente relacionadas a Buffer Overflows(Seção 2.3.1).

- Algumas vulnerabilidades são comuns a várias linguagens, enquanto outras ocorrem em contextos específicos. As vulnerabilidades de *Buffer Overflow* estão diretamente ligadas a linguagens em que não há a proteção de escrita de dados diretamente na memória. As linguagens mais populares que possuem essa característica são, atualmente, C e C++. Desse modo, as análises relacionadas a linguagens são voltadas, neste trabalho, para essas linguagens.

- A análise de ferramentas que implementam essas abordagens de detecção de vulnerabilidades é complexa, necessitando de um conhecimento prévio das abordagens utilizadas, e da utilização de uma metodologia de análise extremamente rígida (de modo a evitar ruídos que comprometam os resultados da análise), entre outros fatores. Essas atividades fogem ao escopo desse trabalho. Serão utilizados, como referência, estudos já realizados.

- O principal objetivo desse trabalho é o estudo do comportamento de metotodologias de análise estática de código, na detecção de *buffer overflow*. O estudo dessas características é feito a partir do levantamento teórico de cada método de análise estática e de uma análise empírica dos resultados de outros estudos realizados, acerca de ferramentas de análise que implementam essas técnicas. vulnerabilidades.

1.7 Trabalhos relacionados

Há uma série de trabalhos já realizados voltados ao estudo e melhoria de ferramentas de análise estática e das técnicas que as implementam. De modo geral, os trabalhos citados ao longo dessa seção se relacioname com o trabalho peloo estudo comparativos de ferramentas de análise estática de vulnerabilidades e/ou estudo das características das técnicas de análise, separadas ou em conjunto.

As subseções à seguir destacam as principais semelhanças e diferenças de foco entre alguns trabalhos relacionados a esta monografia e a mesma.

1.7.1 Trabalhos relacionados ao estudo comparativo de ferramentas

(SHRESTHA, 2013) faz uma análise comparativa de diversas ferramentas *open source* de análise estática de código. Nesse trabalho, foi realizada uma análise teórica de diversas ferramentas que avaliam códigos escritos nas linguagens de programação C, C++ e Java, além de uma observação empírica do desempenho da ferramenta Findbugs (para detecção de vulnerabilidades em código Java). O trabalho de *SHRESTHA* segue a mesma linha do apresentado neste documento. Também é feito um levantamento teórico das técnicas de análise estática implementadas por algumas ferramentas. Além disso, a observação empírica dos resultados de análise da ferremanta Findbugs também é feita utilizando a Suíte de testes do Juliet (mais detalhado na seção 3.1). As diferenças nas linhas de pesquisa adotadas em cada um dos trabalhos se difere principalmente no foco da análise. *SHRESTHA* busca, ao final de seu trabalho, a análise de desempenho da ferramenta em questão utilizando reportes para todas as vulnerabilidades acusadas pela ferramenta. Já no presente trabalho, o principal foco é a observação do desempenho de diversas ferramentas a partir das características de uma dada vulnerabilidade (no contexto desse trabalho, as vulnerabilidades analisadas são as relacionada a *buffer overflow*). Assim, buscou-se, nesse trabalho, uma análise mais específica, relacionando as características de cada técnica com as características das vulnerabilidades que cada ferramenta propõe identificar.

1.7.2 Trabalhos relacionados à junção de técnicas de análise estática

Em um estudo desenvolvido por (RöDIGER, 2011), é apresentada a junção da técnica de análise de fluxo de dados com a técnica de *Model checking* (Checagem de Modelo). Estas e outras técnicas estão melhor caracterizadas na seção 2.2.1.2, mas o trabalho apresentado por ele relaciona-se com o deste documento por buscar meios de integrar técnicas de análise em busca de uma maior eficiência na análise de vulnerabilidades. Röediger busca unir as vantagens dessas duas técnicas de modo a minimizar a fraqueza de cada uma. A checagem de modelo se mostra, segundo o autor, insuficiente para "raciocinar"a respeito do fluxo de execução do programa, mesmo possuindo uma precisão muito boa na detecção detecção de vulnerabilidades, quando comparada com outras técnicas. Já a análise de fluxo de dados possui uma precisão menor, mas é capaz de caracterizar bem o fluxo de execução de um programa a partir dos dados inseridos no mesmo. Assim, o autor procurou utilizar a análise do fluxo de dados como ferramenta para que a técnica de checagem de modelos pudesse se mostrar mais eficiente no entendimento de códigos como um todo e, consequentemente, obter-se um melhor resultado na detecção de vulner-

abilidades. Observa-se, nesse trabalho relacionado, a identificação e uso de características de cada técnica de vulnerabilidade na busca de uma análise mais eficiente. O trabalho apresentado no presente documento se assemelha na busca das características de cada técnica de análise em prol de melhores resultados na detecção de vulnerabilidades. Entretanto, o trabalho apresentado por Rödiger foca em características já bem definidas de cada técnica e é contextualizada pela identificação de vulnerabilidades propagadas pela inserção de dados nos pontos de entrada de dados de um dado programa.

(MUSKE et al., 2013) também utiliza a checagem por modelo para eliminar falsos positivos resultantes de uma interpretação abstrata. O autor define a interpretação abstrata como escalável (aplicável em grande quantidade de dados), mas imprecisa. Já a checagem de modelos é caracterizada como precisa, mas não escalável. Assim, o trabalho apresenta o uso das duas técnicas de maneira conjunta: utilizando-se a interpretação abstrata é possível reduzir fortemente a quantidade de dados com maior probabilidade de ocorrer uma vulnerabilidade. Após isso, a redução de falsos positivos reportados pela interpretação abstrata é obtida pela checagem de modelos, que se mostra mais precisa e que, por possuir uma quantidade de dados menor a ser analisada, consome menos recursos computacionais e ainda conseguiu, nos resultados empíricos apresentados pelo autor, reduzir a quantidade de falsos positivos reportados em 60%. Este trabalho também se relaciona com o apresentado por mim nesse documento por buscar características de cada técnica de análise e utilizá-las de modo a obter benefícios na detecção de vulnerabilidades. Entretanto, as linhas de pesquisa se diferem no foco de cada trabalho. Enquanto Muske parte de características já conhecidas de cada técnica e as aplica em um contexto específico, busco, no trabalho que apresento aqui, identificar características no comportamento de diversas técnicas de análise estática de vulnerabilidade a partir de uma análise empírica. Além disso, características como consumo de recursos não se aplicam ao contexto desse trabalho, visto que, dependendo do modo como a análise é feita, características como consumo de recursos e velocidade de detecção se mostram secundárias.

A convergência de abordagens de análise de vulnerabilidades não fica, necessariamente, restrita a apenas duas. (GROSSO et al., 2008) propõe uma combinação de algoritmos genéticos, programação linear, teste de evolução e informações estáticas e dinâmicas para deteção de *buffer overflows*. A proposta desse trabalho relacionado se baseia na geração automática de dados de entrada, como insumo para definir e ajustar um algoritmo genético na detecção de *buffer overflow*. Esse algoritmo utiliza-se da técnicas supracitadas em seu mecanismo de detecção. Assim como outros trabalhos relacionados descritos nessa seção e o deste documento, Del Grosso buscou integrar diversas técnicas (algumas não necessariamente caracterizadas como técnicas de análise estática, quando isoladas), para aumentar a convergência de algoritmos genéticos na detecção de vulnerabilidades. A principal contribuição desse trabalho relacionado, segundo os autor consiste na automatização de dados de entrada, que são diretamente responsáveis pela eficiência de uma análise feita

com algorimos genéticos.

1.7.3 Trabalhos relacionados a outras abordagens de análise estática

Outros estudos também propõem abordagens diferenciadas para detecção de de vulnerabilidades. (XIE; AIKEN, 2005) apresenta uma possibilidade de junção de duas técnicas para melhoria na detecção de por exemplo, propõe a utilização de satisfação booleana para detecção de erros.

1.8 Organização do documento

Este trabalho está organizado em três partes, sendo estas: Introdução (Parte I), Desenvolvimento (Parte II) e Conclusão (Parte III).

Na Introdução são apresentados uma breve contextualização do tema, uma caracterização do problema proposto e os objetivos que se espera alcançar.

A segunda parte, relativa ao de Desenvolvimento, agrega o conteúdo referente ao referencial teórico utilizado, à metodologia utilizada e aos resultados obtidos. O Referencial Teórico (Capítulo 2) tem por objetivo evidenciar os principais conceitos, termos e definições a serem utilizados neste trabalho. A Metodologia (Capítulo 3) aborda as etapas que utilizadas para alcançar os resultados apresentados nesse trabalho. Os Resultados (Capítulo 4), expõem os dados obtidos no estudo de caso utilizado nesse trabalho, assim como uma análise e discussão a respeito desses resultados, associando-os ao embasamento teórico levantado no Referencial Teórico.

Por fim, a terceira parte, relativa à Conclusão, aborda as considerações mais relevantes relativas aos resultados obtidos e analisados, relacionando-as com os objetivos levantados inicialmente. Essa parte está subdividida em capítulos suscintos de Conclusão e Trabalhos futuros, sendo que o Capítulo 5 apresenta as conclusões nas quais esse trabalho chegou, a partir das questões que orientam essa pesquisa. Já o Capítulo 6 apresenta os Trabalhos Futuros que os resultados e perspectivas obtidas nesse trabalho possibilitam.

Parte II

Desenvolvimento

2 Referencial Teórico

2.1 Segurança de Software

Atualmente, o número de incidências relacionadas a ataques a sistemas de software se mostra cada vez mais recorrente. A exploração de brechas existentes nos softwares, bem como perda de dados, têm gerado um impacto enorme em muitas companhias e organizações, podendo, em casos não tão incomuns, comprometer a própria sobrevivência dessas. Segundo (PAUL, 2008), existem uma série de fatores que, isoladamente ou de maneira combinada, podem provocar as incidências supracitadas. Estas, estão listadas a seguir:

- Proteção insuficiente de dados, na transmissão ou armazenamento dos mesmos;
- Projeto, desenvolvimento e/ou implantação de software feitos de modo inseguro;
- Configuração de controle de segurança de software inapropriada ou inadequada;
- Falhas na transmissão de dados, levando à perda de informações
- Falta de segurança em camadas que circundam os hosts ou aplicações

Em 2006, especialistas da indústria afirmaram que 70% das tentativas de ataque são feitas em nível de aplicação, segundo (AGGARWAL; JALOTE, 2006). Esse fator ocorre em todos os âmbitos, desde aplicações comerciais a projetos *open source*. Devido a isso, há uma grande demanda para detecção de falhas de segurança em códigos-fonte, preferencialmente, no início do ciclo de vida desses softwares.

Ainda que haja essa preferência, (PAUL, 2008) ressalta que a obtenção de um software com alto nível de segurança depende de uma abordagem nesse aspecto em todo o seu ciclo de vida. O desenvolvimento de um código fonte seguro é, sem dúvida, um fator crítico. Mas é preciso um balanceamento de processos, políticas e pessoas, ao longo de todo do ciclo de vida. A Figura (1) mostra o diagrama de um *framework* para incorporar atividades relacionadas à segurança, ao longo de um SDLC (Ciclo de Vida de Desenvolvimento de Software) genérico, evidenciando as atividades básicas a serem incorporadas a cada fase do ciclo, conforme caracterizado por (JONES; RASTOGI, 2004).

Dentre essas fases, destacam-se, para o contexto desse trabalho, as fases de Desenvolvimento e de Teste: a primeira por ser a etapa onde vulnerabilidades, como *buffer overflow*, entre outros são inseridos no código-fonte; a segunda por ser a etapa onde estão

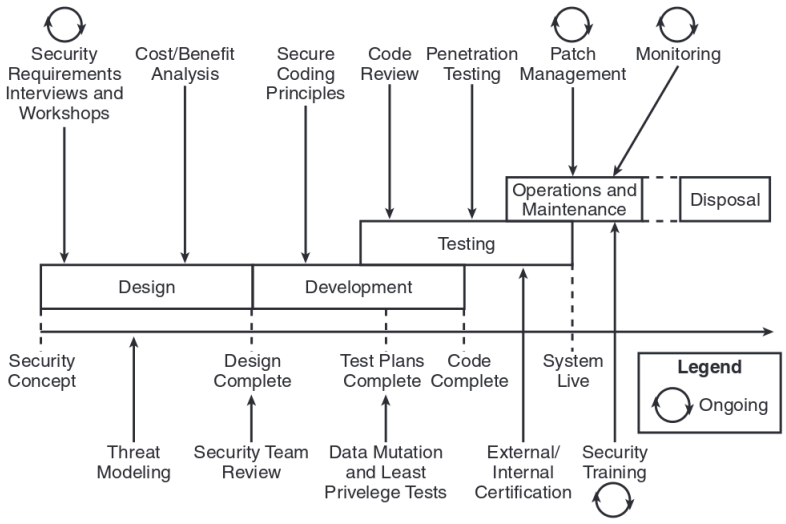

Figura 1 – Segurança no SDLC.

concentrados os esforços para garantir que medidas de contenção foram implementadas corretamente e que o código foi desenvolvido segundos práticas de desenvolvimento seguro.

2.1.1 Fase de Desenvolvimento

Ainda segundo (JONES; RASTOGI, 2004), nesta fase, devem ser adotadas práticas relativas à codificação segura, como as exemplificadas abaixo.

- **Validação de dados:** Sempre validar todos os dados de entrada do sistema, sejam entradas dadas pelo usuário, variáveis de ambiente, arquivos de entrada, entre outros;
- **Ambiente hostil:** Sempre partir do princípio que o sistema operará em um ambiente hostil, interna e externamente;
- **Padrões abertos:** Utilizar padrões abertos para desenvolvimento, visto que os mesmos puderam ser avaliados por toda a comunidade de desenvolvimento e são, portanto, mais confiáveis;
- **Componentes confiáveis:** Minimizar e proteger a interação entre os componentes do sistema, visto que a interdependência destes representa um risco para este sistema;
- **Dados em processo, transmissão e armazenamento:** Ter medidas de segurança para proteger os dados em qualquer estado em que se encontrem.
- **Autenticação:** Nunca permitir o acesso, de usuário ou de programação, sem a autenticação apropriada;

- **Proteções de segurança nativas:** Não subverter medidas de segurança já existentes no ambiente em que o sistema se encontra;

- **Falhar de forma segura:** Garantir que se a aplicação falhar, ela não abrirá brechas de segurança;

- **Registro, monitoramento e Auditoria:** Sempre registrar e monitorar o que está acontecendo;

- **Data e Hora:** Usar um sistema de data e hora preciso;

- **Privilégio mínimo:** Trabalhar sempre com o privilégio mínimo necessário para executar a ação desejada;

- **Tolerância a Faltas:** Tratar exceções apropriadamente, em nível de código, evidenciando as informações necessárias sobre a exceção, para fins de correção;

- **Criptografia forte:** Usar padrões de cifração reconhecidamente fortes;

- **Randomicidade:** Ao utilizar números randômicos, eles devem ser não determinísticos.

Além disso, também é preciso adotar medidas de segurança que combatam problemas de vulnerabilidades de código como *buffer overflow*, problemas de *deadlock*, controle de granularidade, validação de parâmetros, entre outros.

2.1.2 Fase de Testes

Nesta fase, deve ser desenvolvido um plano de testes de segurança que leve em consideração os riscos já levantados em outras etapas (neste caso, a fase de Avaliação de riscos se destaca).

A fase de Testes possui quatro atividades principais: teste de unidades, teste de garantia de integração e qualidade, teste de penetração e certificação. É na atividade de testes de unidades, que são utilizadas ferramentas de análise estática e dinâmica de código, para detecção de vulnerabilidades no sistema. Embora as outras fases de um SDLC devam ter tanta atenção e esforço quanto as supracitadas, abordam-se aqui, apenas duas, de modo a elucidar o contexto da necessidade de utilização de ferramentas de análise de código, para detecção de vulnerabilidades de código.

2.2 Sobre a análise de Código

2.2.1 Sobre a análise Estática de Código

Análise estática de código é o processo de avaliação de um sistema ou componente baseado na sua forma, estrutura, conteúdo ou documentação. Da perspectiva de análise de software, a análise estática permite localizar vulnerabilidades no código de programas, que podem ocasionar em uma invasão, comportamento não esperado, ou até mesmo falha desses programas. Ela pode ser feita de forma manual em contextos muito específicos, como a inspeção de um determinado trecho código. Mas na grande maioria das vezes, essa análise é feita de forma automatizada através do uso de uma ou mais ferramentas de análise.

Essa análise estática pode pode ocorrer em cooperação com um compilador, de maneira independente através da análise do código-fonte, ou ou mesmo após a compilação de um determinada código fazendo, nesse caso, uma checagem do *byte code* e do código binário, como afirma (TEIXEIRA, 2007). Esse último tipo de abordagem pode ser muito útil quando o código fonte do software em questão não está disponível.

2.2.1.1 Alguns tipos de ferramentas de análise estática

A análise estática de software abrange caracterísitcas e objetivos amplos, sendo a análise voltada para a detecção de vulnerabilidades apenas um dos tipos de análise estática. De modo geral, as ferramentas de análise estática pode ser divididas em várias categorias, exemplificadas por (CHESS; WEST, 2007). Cada tipo, possui intuito e características específicas. Cita-se aqui: Checagem de tipo, Checagem de estilo, Entendimento do programa, Verificação do programa, Checagem de propriedades, Descoberta de erros e Revisão de segurança.

Checagem de tipo

Consiste em uma das formas de análise estática mais utilizadas, sendo voltado para checagem dos tipos das variáveis, tanto em suas declarações quanto em suas atribuições. Deste modo, muitos erros acidentais podem ser evitados, como atribuições de valores primitivos (inteiros, por exemplo) a objetos, em Linguagens Orientadas a Objeto. Outra característica abordada por esse tipo de checagem é a conversão de tipos de variáveis, em que pode ocorrer a perda de dados.

Checagem de estilo

A checagem de estilo é uma abordagem mais delicada e superficial que a checagem de tipo. Ela se caracteriza por checar características relacionadas a espaços em branco, nomenclaturas, funções depreciadas, estrutura do programa, entre outros. Os erros levantados por esse tipo de abordagem geralmente não influem na execução correta do pro-

grama, mas afetam a legibilidade e manutenabilidade do mesmo.

Entendimento do programa

As ferramentas que possuem esse tipo de abordagem são voltadas para indexação das funcionalidades de programas, possibilitando entender e analisar o comportamento dos mesmos. Esse entendimento engloba características como declaração e uso de métodos e variáveis, por exemplo. A partir dessa avaliação é possível associar o comportamento de programas em alto nível, com o seu código em si.

Verificação do programa

As ferramentas com esse tipo de abordagem buscam avaliar se o código do programa atende às especificações de implementação e comportamento previamente estabelecidos. A principal limitação desse tipo de abordagem encontra-se na complexidade de se caracterizar, satisfatoriamente, as especificações do programa.

Checagem de propriedades

Semelhante aos princípios da abordagem anterior, a checagem de propriedades consiste na verificação das características do código a partir de especificações parciais, que detalham apenas parte do comportamento do programa. A maior parte das ferramentas com esse tipo de abordagem, aplicam inferência lógica ou checagem de modelos.

Descoberta de erros

As ferramentas com esse tipo de abordagem objetivam localizar pontos do código onde algo pode ter sido implementado erradamente de maneira acidental. Certas instruções que, a princípio, não causariam erro, podem resultar em um comportamento inesperado, dependendo do compilador ou interpretador do programa. Ferramentas que implementam essa abordagem de maneira mais avançada conseguem, por exemplo, identificar padrões de instruções implementados no código e alertar mudanças nesse padrão.

Revisão de segurança

Ferramentas com essa abordagem utilizam-se de técnicas encontradas em outras ferramentas, mas implementadas de modo a identificar problemas de segurança.

2.2.1.2 Técnicas para detecção de vulnerabilidades

As técnicas de análise estática a seguir visam a detecção de possíveis vulnerabilidades a partir da interpretação sintática e/ou semântica de um dado código. Muitas ferramentas implementam duas ou mais técnicas, em conjunto, para uma análise mais precisa das vulnerabilidades.

Através de técnicas e abordagens de análise estática, e da detecção de padrões de vulnerabilidades, como as caracterizadas em CWEs (Seção (2.3)), para a análise de *buffer*

overflows, por exemplo, é possível obter informações a respeito do comportamento do programa analisado e identificar possíveis vulnerabilidades de código presentes no mesmo. Devido à grande variedade de características relativas às vulnerabilidades de software, bem como a própria variação de complexidade na detecção de uma CWE específica, a eficiência de uma ferramenta de análise estática está condicionada ao modo como foi implementada, não podendo, assim, identificar vulnerabilidades de código ou características para as quais não foi programada. Este fator pode ocasionar na exploração das limitações da ferramenta, por parte de alguém mal intencionado, por exemplo.

Entre possíveis implementações de ferramentas de análise estática de código, pode-se citar (ALMORSY; GRUNDY; IBRAHIM, 2012), que utiliza as características de determinadas vulnerabilidades, como assinaturas dessas vulnerabilidades para identificação de ocorrências dessas, através da análise estática.

2.2.1.2.1 Análise Léxica ou Detecção de Padrões

Essa técnica se baseia na criação de *tokens* (ou bastões), a partir do código a ser analisado, de modo a descartar linhas em branco, comentários ou outros elementos desnecessários para a compreensão semântica do programa. Para identificação desses *tokens*, utiliza-se expressões regulares. Além de um código identificador, esses tokens podem possuir informações como sua posição no código original, possibilitando um rastreamento reverso para localização de uma possível vulnerabilidade.

2.2.1.2.2 Análise de Fluxo de Dados

Essa técnica consiste na exploração dos diferentes caminhos de execução possíveis a partir de uma função ou estrutura de decisão. A partir dessa análise, é possível construir, entre outras coisas, um grafo de chamadas, representando potenciais fluxos entre funções ou métodos. A partir da identificação desses fluxos, é possível avaliar semanticamente o código analisado. De maneira intuitiva, é o modo que utilizamos, manualmente, para determinar se há ou não uma vulnerabilidade em um determinado trecho de código. Essa técnica é feita percorrendo o código a partir do início ou de um determinado trecho e seguindo, através das estruturas condicionais, laços e chamadas de função, os fluxos de execução possíveis. No trecho de código escrito em linguagem C, a seguir, é possível detectar a ocorrência de um *buffer overflow* seguindo o fluxo executado pelo algoritmo.

Listing 2.1 – Código que resulta em um Buffer Overflow

```
1 void f()
2 {
3     int x = 1;
4     int y = 1;
```

```
5       char a[10];
6       if ((x + y) == 2)
7       {
8           a[20] = 0;
9       }
10  }
```

Nesse trecho, uma ferramenta que não faça uma análise do fluxo dos dados das variáveis é incapaz de determinar em que circunstâncias o programa executará a instrução da linha 6, que caracteriza um tipo de *buffer overlflow*. Assim, uma alternativa de ação desta ferramenta seria o reporte incondicional da ocorrência. Entretanto, caso a condição existente na linha 4 seja um tratamento que evite a essa falha, o reporte dessa ocorrência caracterizará um falso positivo[1].

Do mesmo modo, a representação do índice de acesso à variável a, na linha 6 do trecho de código a seguir, através de uma variável recebida como parâmetro da função f(), resulta em uma situação que uma inspeção sem análise de fluxo de dados é incapaz de determinar se isso resultará ou não em um *Buffer Overflow*. Uma ferramenta que não implemente essa técnica, tende a não reportar a occorrência de um *Buffer Overflow*, independentemente do valor que a variável *size* receba.

Listing 2.2 – Código que pode resultar ou não em um *Buffer Overflow*

```
1   void f(int size)
2   {
3       int x = 1;
4       int y = 1;
5       char a[10];
6       if ((x + y) == 2)
7       {
8           a[size] = 0;
9       }
10  }
```

2.2.1.2.3 Shape Analysis

É uma forma de análise de ponteiros relativamente mais precisa que outras análises do mesmo tipo. A técnica de análise de ponteiros busca determinar o conjunto de objetos os quais um dado ponteiro pode refernciar. Essas técnicas trabalham, necessariamente, por aproximação. Nesse contexto, a *Shape Analysis* pode determinar, de forma um pouco mais precisa, esse conjunto de objetos.

[1] Pontuação de Resultados de Ferramentas (Seção 3.2)

2.2.1.2.4 Lógica de Separação

Lógica de separação, é uma extensão da lógica de Hoare(HOARE, 1969), sendo um método formal para raciocinar sobre programas que acessam e modificam dados mantidos na memória do computador. Ele é baseado na separação através de conjuntos P * Q, que afirma que P e Q espera por porções separadas de memória, e em regras de prova de programas que exploram a separação de fundamentação modular sobre esses conjuntos(O'HEARN, 2012).

2.2.1.2.5 Interpretação Abstrata

É uma técncia formalizada por Cousot e Cousot para abstrair aspectos que não são relevantes para a interpretação desejada, de modo a executar uma interpretação mais enxuta e eficiente dos aspectos restantes (COUSOT; COUSOT, 1977).

2.2.1.2.6 Checagem de Modelos

Considerando que um algoritmo implementado trabalha de acordo com limitações de memória, pode-se transformar um algoritmo em uma máquina de estados finitos. A checagem de modelos consiste em transformar o algoritmo a ser checado em um máquina de estados finitos (chamado modelo), e então comparar a especificação com um modelo de checagem. Através dele é possível analisar, por exemplo, se um trecho de memória não está sendo desalocado duas vezes, o que resultaria em um erro, ou mesmo se um ponteiro não nulo não está sendo mais referenciado.

2.2.1.3 Capacidades e Limitações

A análise estática possui limitações resultantes da não execução do algoritmo analisado. Inferir sobre o comportamento de um determinado algoritmo pode resultar em conclusões errôneas, gerando falsos positivos e/ou falsos negativos.

Quando a análise é feita em algoritmos em linguagem natural, adiciona-se a complexidade de determinar o local específico da falha. Assim, diferentes ferramentas podem apresentar resultados de maneira distinta para uma mesma vulnerabilidade detectada.

A eficiência de se inferir sobre o comportamento de um algoritmo está diretamente relacionada com a simplicidade desse algoritmo. Aumentando-se a complexidade dos possíveis caminhos e estados de um programa, aumenta-se a complexidade de busca de vulnerabilidades através da análise estática.

2.2.2 Sobre a análise Dinâmica de Código

A análise dinâmica é feita a partir do comportamento da aplicação em tempo de execução. Segundo (EADDY et al., 2008), uma análise de *tracing* de execução baseada em localização, por exemplo, analisa o comportamento de execução de um determinado programa para determinar elementos que foram ativados quando um comportamento desejado é executado. Por *ativado*, entende-se que houve a execução (no caso de funções e métodos), ou o acesso de leitura e/ou escrita (no caso parâmetros, campos e variáveis globais). Para comportamento analisado, são obtidas uma série de registros relativos a cada *ativação*. A partir da análise e comparação das ativações feitas em um dado conjunto de comportamentos escolhidos, é possível distinguir elementos que são específicos de uma característica do programa.

Uma das principais desvantagens desse tipo de análise, reside no fato de que o ambiente de teste pode não propiciar todas as entradas possíveis para um determinado contexto, podendo assim, gerar falsos negativos (não abordando possíveis estados em que a vulnerabilidade se manifestaria). Soma-se a isso a dificuldade de se replicar uma mesma eficiência de análise em diferentes contextos, visto que a análise está diretamente ligada à execução do algoritmo que está sendo analisado.

2.3 CWEs

CWE (Common Weakness Enumeration) é uma lista, ou dicionário, formal desenvolvido por pessoas das comunidades acadêmica, do setor comercial e governamental. Ela tem por objetivo unir, da melhor forma possível, abrangência e profundidade de conteúdo, no que se refere à área de segurança de software, de modo a servir como uma definição padrão. As principais características dessa lista são caracterizadas abaixo:

- Serve como uma linguagem padrão para descrição de falhas de segurança de arquitetura, design ou código-fonte de softwares;
- Serve como padrão de medição para ferramentas de segurança de software voltadas à identificação dessas vulnerabilidades;
- Fornece uma definição de base comum para esforços de identificação, mitigação e prevenção de vulnerabilidades de código;
- É aprovada pela indústria, através da comunidade CWE e produtos CWE-compatíveis.

Entre os tipos de falha de software presentes nesta lista, pode-se citar: *buffer overflow*, problemas de validação de segurança, erros de caminhos e/ou canais, manipulação de erros, entre outros. Segundo (THE MITRE CORPORATION, 2014), estão catalogadas 943 CWEs, incluindo fraquezas, categorias, elementos compostos e perspectivas.

Como perspectiva, por exemplo, pode caracterizar uma CWE: como pertencente a um conjunto específico de linguagens de programação, plataformas ou tecnologias; segundo o risco que ela implica; segundo uma classificação ou taxonomia, entre outros. Como fator comum a todas as CWEs catalogadas, têm-se como características de uma CWE:

- Identificador da CWE / Nome do tipo de fraqueza
- Descrição do tipo de CWE
- Nomes alternativos para a fraqueza
- Descrição do comportamento da fraqueza
- Descrição das formas de exploração (ataque) da fraqueza
- Eficiência de formas de exploração (ataque) da fraqueza
- Possíveis mitigações
- Informações sobre CWEs relacionadas
- Estrutura característica do código
- Exemplos de código para linguagens/arquiteturas
- Identificador CVE[2] do número de fraqueza para os quais a fraqueza existe
- Referências

2.3.1 CWEs Diretamente relacionadas a *Buffer Overflows*

Os problemas relacionados a *buffer overflow* estão relacionados a mais de 50 % dos problemas de segurança, segundo (JONES; RASTOGI, 2004), sendo um dos mais importantes tipos de erro relacionados à linguagens em que não há proteção de memória, como a linguagem C, por exemplo. Há ainda reportes mais atuais como (CHRISTEY, 2011), que destaca a Cópia de Buffer sem Checagem do Tamanho de Entrada (*'Classic Buffer Overflow'*) no Top 3, dos 25 erros de software mais perigosos.

(KRATKIEWICZ; LIPPMANN, 2005) caracteriza a ocorrência de um *buffer overflow* quando dados podem ser escritos fora do local de memória alocado para um dado *buffer*, seja após o fim deste, seja antes do início do mesmo. Há diversas variações de *buffer overflow*, podendo ocorrer na pilha de armazenamento de variáveis locais (*stack*), na área de alocação dinâmica de dados (*heap*), em segmentos de dados, ou no BSS (área de memória usada por um programa para alocação de dados globais não inicializados). Aliado a isso, a estrutura de um *buffer overflow* pode ser descrita através da caracterização de 22 atributos, conforme listado na tab (2.3.1).

[2] https://cve.mitre.org/

Numero do Atributo	Nome do Atributo
1	Leitura/Escrita
2	Limite Inicial/Final
3	Tipo de Dado
4	Local da Memória
5	Escopo
6	Recipiente
7	Ponteiro
8	Complexidade de Indexação
9	Complexidade de Endereçamento
10	Complexidade do Tamanho/Limite
11	*Alias* do Endereço do Buffer
12	*Alias* do Índice do Buffer
13	Fluxo de Controle Local
14	Fluxo de Controle Secundário
15	Estrutura de Laço de Repetição
16	Complexidade de Laço de Repetição
17	Assincronia
18	"Manchas"
19	Dependência do Ambiente de Execução
20	Magnitude
21	Discreto/Contínuo
22	Erro de Sinalização/Não Sinalização

Tabela 1 – Atributos da Taxonomia de um *Buffer Overflow*

2.3.1.1 Subgrupos de *Buffer Overflow*

2.3.1.1.1 Leitura/Escrita:

Descreve o tipo de acesso de memória. Um acesso com permissão de escrita pode ser explorado em ataques típicos de *buffer overflow*, enquanto um acesso com permissão de leitura pode permitir, no mínimo, o acesso não autorizado ao conteúdo de determinadas áreas de memória.

As linguagens caracterizam, de maneira diferente, o modo de acesso para leitura ou para escrita. Assim, é possível que uma ferramenta de análise estática avalie de maneira diferente a ocorrência de um *buffer overflow* em um acesso de leitura em relação a um acesso de escrita.

2.3.1.1.2 Limite Inicial/Final:

Descreve a borda do espaço de memória alocado que foi violada no *buffer overflow*. Na linguagem C, por exemplo, se um ponteiro estive apontado para o início de um *buffer* e for chamado passando como referência um índice negativo, certamente ocorrerá um acesso

fora dos limites do buffer. Entretranto, a premissa de um índice negativo não se aplica caso o ponteiro esteja apontando para outra posição do *buffer*.

Ainda que não seja uma prática usual, é possível até mesmo que um ponteiro esteja apontando, inicialmente, para um local fora dos limites do *buffer*. Nesse caso, um *buffer overflow* não ocorrerá se a aritmética de uso do ponteiro for utilizada de tal forma que o acesso seja feito dentro dos limites do *buffer*.

2.3.1.1.3 Tipo de Dado:

Indica o tipo de dado alocado na região do buffer. Variações do tipo de dado previamente alocado, com o tipo de dado a ser escrito ou lido, pode resultar em um *buffer overflow*.

2.3.1.1.4 Local da Memória:

Indica o tipo de local onde o buffer se encontra. A memória pode ser dividida em dois tipos: *Stack* e *Heap*. A *Stack* (Pilha) é onde são guardadas as variáveis locais, nos programas. Já a *Heap* (memória estática) é onde os valores armazenados permanecem até o fim do programa. Assim, *buffers* de dados armazenados em *heap* têm seu tamanho fixo do início ao fim do programa. Já *buffers* armazenados em *stack* podem ter seu tamanho modificado em tempo de execução, além de terem sua alocação e desalocação como responsabilidade do programador.

2.3.1.1.5 Escopo:

Descreve a diferença entre o escopo no qual o *buffer* foi alocado, e o escopo no qual foi executado. Dependendo a forma de análise, a não consideração desse fator pode ocasionar em uma não detecção da vulnerabilidade de código.

2.3.1.1.6 Recipiente:

Indica se o *buffer* está contido em algum tipo de recipiente (*Container*).

2.3.1.1.7 Ponteiro:

Indica se o acesso ao *buffer* é feito utilizando um ponteiro de referência. Isso caracteriza a possibilidade de se utilizar um ponteiro de acesso sem a presença de um indexador, por exemplo.

2.3.1.1.8 Complexidade de Indexação:

Indica a complexidade de indexação do *buffer*. A indexação pode ser constante, variável, uma expressão linear ou não linear, retorno de uma função ou mesmo o conteúdo de outro *buffer*.

2.3.1.1.9 Complexidade de Endereçamento:

Descreve a complexidade de cálculo do endereço desejado, por parte do ponteiro (sendo relativo ao modo de indexação já citado).

2.3.1.1.10 Complexidade do Tamanho/Limite:

Indica a complexidade do tamanho ou limite passado como parâmetro de uma função que pode ocasionar um *buffer overflow*. Ressalta-se que algumas funções podem ocasionar um *buffer overflow* sem que esse fator esteja envolvido (ex: *strcpy*), por não receberem o tamanho ou limite como parâmetro.

2.3.1.1.11 *Alias* do Endereço do *Buffer*:

Indica se o *buffer* é acessado diretamente ou através de *alias* (apelidos). Releva-se aqui, o fato de que o acesso ao buffer pode ser feito de maneira indireta. Um exemplo seria o acesso através de um ponteiro que referencie outro que, por sua vez, referencie a região de memória em questão. Caso o acesso seja feito de maneira direta, esse atributo não é aplicável.

2.3.1.1.12 *Alias* do Índice do Buffer:

Indica se o índice do *buffer* é, referenciado através de *alias*, de modo similar ao atributo anterior. Vale ressaltar que esse tipo de atributo é um dos fatores que dificultam a detecção de *buffer overflows* por ferramentas de análise de código.

2.3.1.1.13 Fluxo de Controle Local:

Descreve o modo como o fluxo de controle local afeta ou influencia o *overflow*. Para isso, são avaliados elementos como *if/else, switch, goto*, ponteiro recursivo, entre outros.

2.3.1.1.14 Fluxo de Controle Secundário:

Possui características semelhantes ao Fluxo de Controle Local, diferenciando-se apenas no local onde o mesmo os dois são construídos. O Fluxo de Controle Local sempre precede ou contém o Secundário, devendo ser analisado apropriadamente, segundo esse contexto.

2.3.1.1.15 Estrutura de Laço de Repetição:

Descreve o tipo de laço de repetição onde o *overflow* ocorre, se houver um laço. Este pode ser um laço *for, do-while, while* padrão ou não-padrão. Caracteriza-se por padrão, o laço onde há uma inicialização, um teste de saída e uma variável de incremento ou decremento para controle do laço (todos em seus formatos e locais típicos).

2.3.1.1.16 Complexidade de Laço de Repetição:

Indica a quantidade de componentes (no que se refere a inicialização, teste de saída e incremento/decremento), que é mais complexa do que o padrão (comum). A avaliação que se faz em relação a esse atributo se deve à dificuldade com que ferramentas de análise conseguirão identificar um *buffer overflow* nessas circunstâncias.

2.3.1.1.17 Assincronia:

Indica se a possível ocorrência de *buffer overflow* pode ou não ser influenciada por manipuladores de execução assíncronos, como *threads, forked processes*, ou *signals*. Como esses manipuladores são, até certo ponto, dependentes do Sistema Operacional para o qual o software foi projetado, uma possível ferramenta de análise de código deve levar em conta a sintaxe utilizada para o ambiente em questão.

2.3.1.1.18 'Manchas':

Descreve se e como um *buffer overflow* pode ser influenciado por um fator externo, como variáveis de ambiente, argumentos *argc, argv*, leitura de arquivos ou de entrada padrão, sockets ou ambiente de processo. Não se pode assegurar a presença de uma ocorrência de *buffer overflow* sem que se analise o contexto dessa ocorrência, pois pode ser este o fator determinante para a ocorrência, ou não, dessa vulnerabilidade de código.

2.3.1.1.19 Dependência do Ambiente de Execução:

Indica a dependência ou não do ambiente de execução, para a ocorrência da vulnerabilidade de código. O uso de valores gerados por um gerador randômico é um exemplo dessa dependência.

2.3.1.1.20 Magnitude:

Indica o tamanho do *overflow*, em bytes. Esse atributo está diretamente ligado ao modo como a ferramenta age na detecção do *buffer overflow*.

2.3.1.1.21 Discreto/Contínuo:

Verifica se o local onde ocorre o *buffer overflow* é endereçado arbitrariamente (discreto) ou se o estouro ocorre logo após o limite inferior ou superior do *buffer* (contínuo). Algumas ferramentas dependem dos limites do *buffer* para realizar uma análise, sendo eficientes para detecção de um *buffer overflow* contínuo, em detrimento da detecção de *overflows* discretos.

2.3.1.1.22 Erro de Sinalização/Não Sinalização:

Indica se o *buffer overflow* ocorreu em decorrência do uso de um valor sinalizado ou não sinalizado, sendo relevado aqui, a ocorrência de um valor diferente do que era esperado.

Dadas as características aqui apresentadas, com base em (THE MITRE CORPORATION, 2014), utiliza-se como objeto de estudo desse trabalho, a análise da CWEs relacionadas à CWE-120: Cópia de Buffer sem Checagem do Tamanho de Entrada. O Anexo A caracteriza o relacionamento entre as CWEs relacionadas a *Buffer Overflow*. Destas, as mais diretamente ligadas à CWE-120 são caracterizadas pelo tipo de armazenamento em memória (*stack* ou *heap*), pelo tipo de acesso (leitura ou escrita) e pelo limite do *buffer* (limite inferior ou superior). Assim, destacam-se como objeto de estudo nesse trabalho as CWEs:

- CWE121: *Stack Based Buffer Overflow*[3]
- CWE122: *Heap Based Buffer Overflow*[4]
- CWE124: *Buffer Underwrite*[5]
- CWE126: *Buffer Overread*[6]
- CWE127: *Buffer Underread*[7]

[3] http://cwe.mitre.org/data/definitions/121.html
[4] http://cwe.mitre.org/data/definitions/122.html
[5] http://cwe.mitre.org/data/definitions/124.html
[6] http://cwe.mitre.org/data/definitions/126.html
[7] http://cwe.mitre.org/data/definitions/127.html

3 Metodologia de Análise de Ferramentas

A obtenção de dados de avaliação e melhoria de ferramentas de análise de vulnerabilidades se mostra extremamente complexa. Isso decorre da própria definição abstrata de cada vulnerabilidade a ser identificada. Essa abstração dificulta fortemente a formação de um padrão de caracterização e de identificação dessas vulnerabilidades. As CWEs vêm como um grande passo rumo a essa padronização. Ainda assim, a implementação de técnicas de detecção de forma abrangente, no que tange à variedade de CWEs, é praticamente inviável, pela própria natureza diversificada dessas vulnerabilidades. Assim, o processo de avaliação de ferramentas desse tipo de análise também se torna complexo.

Sem uma descrição formal e precisa das características e possibilidades de ocorrência de uma dada vulnerabilidade, a avaliação de ferramentas que investiguem essas ocorrências se torna extremamente subjetiva. A necessidade de padronização e formalização dos métodos de análise se faz ainda mais necessária para esse contexto de análise de código.

Assim, (CAS, 2011) propõe uma metodologia de estudo de ferramentas de análise estática de código da forma mais objetiva possível. Dentre as características da metodologia apresentada, destacam-se as seções a seguir.

3.1 Casos de Teste

A avaliação da eficiência de uma dada ferramenta de análise estática de código, se dá através da verificação dos resultados de uma ou mais análises feitas em um software. Este (aqui caracterizado por um conjunto de um ou mais arquivos de código que possam ser analisados) precisa ter suas principais características conhecidas para que a eficiência da ferramenta avaliada possa ser analisada. Esse conjunto de dados pode ser natural ou artificial.

Um conjunto de dados de um software natural é proveniente de um software que não foi construído especificamente para a avaliação de ferramentas de análise. Cita-se, como softwares desse tipo, aplicações *open-source* como navegadores, web-servers, e sitemas operacionais, entre outros. A vantagem da utilização desse tipo de dados se dá pela maior facilidade em assimilar os resultados com a realidade, visto que a fonte de dados também é real. Assim, a frequência de ocorrência de determinadas vulnerabilidades se aproxima da real, dependendo das características da fonte utilizada.

Já um software artifical, nesse contexto, caracteriza-se por ser construído especificamente para a avaliação de ferramentas de análise de vulnerabilidade de código. Esse tipo

de conjunto de dados possui vantagens em relação a uma fonte natural por ser mais facilmente caracterizável dentro dos parâmetros desejados para a análise. Entretanto, é preciso ressaltar que a frequência de ocorrência de determinadas vulnerabilidades se apresenta de maneira totalmente distinta da frequência real.

Atualmente, uma das fontes artificiais de dados para testes mais utilizada é o conjunto de casos de teste criado pela *Center for Assured Software* da NSA (Agência de Segurança Nacional Americana), chamada ***Juliet Test Suites***[1].

3.2 Pontuação de Resultados de Ferramentas

Cada caso de teste do Juliet foi desenvolvido para representar uma CWE e contém blocos de *bad code* (vulnerável) e *good code* (onde a vulnerabilidade foi tratada e não se manifestaria na execução do código). A partir do contexto de cada caso de testes e os reportes das ferramentas associados a uma CWE pertencentes ao mesmo grupo do caso de testes analisado, o resultado é classificado segundo as premissas caracterizadas na tabela (3.2).

Tipo de Resultado	Característica
Verdadeiro Positivo (TP)	A ferramenta reporta corretamente a falha alvo do caso de teste.
Falso Positivo (FP)	A ferramenta reporta uma falha do tipo alvo no caso de teste, mas em um código seguro.
Falso Negativo (FN)	A ferramenta não reporta a falha alvo em um caso de teste onde a falha existe.
Indefinido (blank)	As situações que não se encaixam nas características acima. Assim, ou a ferramenta não está em um caso de testes, ou a falha reportada não está associada ao caso de testes.

Tabela 2 – Atributos da Taxonomia de um *Buffer Overflow*

Ressalta-se aqui, que é preciso associar o *report* dado pela ferramenta, com a caracterização da vulnerabilidade existente em cada caso de teste. A análise de uma ferramenta no Juliet, por exemplo, precisa ser feita associando-se o reporte dado pela ferramenta com as CWEs que caracterizam os casos de teste dessa suíte.

Um modo de associar uma dada ferramenta às CWEs, para permitir a análise sobre a suíte de testes do Juliet, é o agrupamento de Classes de vulnerabilidades. Entre as categorias de *weakness* classificadas por (CAS, 2011), cita-se (a título de informação): *Buffer overflows*, Erros numéricos, Perda de informação, validação incorreta de entrada,

[1] Esta suíte de testes está disponível publicamente através do NIST em http://samate.nist.gov/SRD/testsuite.php

entre outras. Neste trabalho, são agrupadas as principais CWEs relacionadas a *Buffer Overflow*.

3.3 Métricas

Após a obtenção dos resultados da análise de uma determinada ferramenta, é possível calcular métricas específicas para esse tipo de análise. (CAS, 2011) descreve as principais métricas que podem ser obtidas a partir dos resultados reportados pela avaliação das ferramentas executadas.

3.3.1 *False Positive Rate* (Taxa de Falsos positivos)

Evidencia a taxa de falsos positivos representada pelo total de falsos positivos retornados pela ferramenta, sobre o total de positivos reportados, verdadeiros ou não. Essa taxa é dada pela razão que segue.

$$FalsePositiveRate = \left(\frac{FP}{TP + FP}\right) \tag{3.1}$$

3.3.2 *Precision* (Precisão)

A precisão, ou taxa de verdadeiros positivos (também conhecida como valor preditivo positivo), é calculada como a razão entre o total de verddeiros positivos reportados pela ferrameta e o total de positivos falsos e verdadeiros, como evidenciado na fórmula a seguir.

$$Precision = \left(\frac{TP}{TP + FP}\right) \tag{3.2}$$

Observa-se a relação entre a precisão e a taxa de falsos positivos, sendo uma métrica o complemento da outra. Ressalta-se também a necessidade de se diferenciar Precisão e Acurácia. A primeira caracteriza o quão bem uma ferramenta identifica pontos falhos (verdadeiros positivos), enquanto a segunda o quão bem uma ferramenta identifica pontos falhos (verdadeiros positivos), e pontos não falhos. Além disso, observa-se que a precisão varia entre 0 (caso reporte apenas falsos positivos) e 1 (caso reporte apenas verdadeiros positivos). Caso não seja relatada nenhuma falha (TP ou FP), a precisão é indefinida. Assim, essa métrica permite entender o quão confiável é o informe de uma ocorrência de fraqueza de código por parte de uma ferramenta.

3.3.3 *Recall*

Representa a fração de vulnerabilidades reportadas pela ferramenta em relação à quantidade real de falhas existentes. É calculada a partir da razão entre os verdadeiros positivos e a soma dos verdadeiros positivos com os falsos negativos conforme a fórmula a seguir.

$$Recall = \left(\frac{TP}{TP + FN}\right) \tag{3.3}$$

Assim como a Precisão, essa métrica resulta em um valor entre 0 e 1, inclusive. Quanto mais próximo de 1 for o valor obtido, mais eficientemente a ferramenta identifica as fraquezas existentes no código analisado. Em contrapartida, valores mais baixos evidenciam um alto número de falsos negativos reportados pela ferramenta.

3.3.4 *F-Score*

Representa a média harmônica entre *Precision* e *Recall*. Essa métrica auxilia na na identificação de uma boa ferramenta ponderando quantas fraquezas são encontradas (verdadeiros positivos) e quanto de ruído é produzido (falsos positivos). Uma ferramenta que tenha uma precisão alta, mas que possua um Recall baixo, ou mesmo o inverso, não é considerada melhor que obtenha valores medianos dessa métricas. De modo geral, a partir da média harmônica é possível classificar cada ferramenta de maneira mais balanceada.

Seu valor é calculado a partir da fórmula a seguir.

$$FScore = 2 * \left(\frac{Precision * Recall}{Precision + Recall}\right) \tag{3.4}$$

3.3.5 *Discrimination* (Discriminação) e *Discrimination Rate* (Taxa de Discriminação)

Discriminação e taxa de discriminação são usadas para determinar se uma ferramenta consegue discernir entre um código ruim e um código bom. Essa métrica se aplica somente às análises em que são utilizados softwares artificiais, como o Juliet. Em softwares assim, cada caso de teste é composto por dois trechos de código: um trecho de *bad code* (código ruim), onde há a vulnerabilidade para a qual o caso de teste se aplica (Verdadeiro Positivo), e um trecho de *good code* (código bom), conde a vulnerabilidade em questão é tratada e, portanto, não deve ser reportada por qualquer ferramenta de análise (se reportado, deverá ser considerado um falso positivo).

A discriminação é contabilizada para cada caso de teste em que um Verdadeiro Positivo é reportado e, no mesmo caso de teste, um trecho em que a vulnerabilidade foi

tratada não é reportado. (CAS, 2011) define como propósito dessas duas métricas a diferenciação entre ferramentas com padrões simplificados de identificação de vulnerabilidades e ferramentas com análise mais complexa. Uma ferramenta que implemente uma detecção de padrões simples (como a Análise Léxica(Seção 2.2.1.2.1), por exemplo), pode reportar uma grande quantidade de ocorrências verdadeiras. Mas a simplicidade com que caracteriza uma falha pode resultar em um alto índice de falsos positivos, o que é evidenciado por uma baixa taxa de discriminação.

3.3.5.1 *Discrimination* (Discriminação)

O número de discriminações creditados a uma dada ferramenta é calculado a partir da quantidade de vezes em que a mesma reporta uma falha (TP) no trecho de código com falha, em cada caso de testes, sem que a ferramenta reporte uma falha no trecho de código sem falhas (ou seja, sem reportar um falso positivo).

O método de obtenção dessa métrica se diferencia das demais por precisar ser analisado a partir de cada caso de testes individualmente. Assim, não se pode calculá-la a partir dos resulados (TP, FP E FN) obtidos de um conjunto e casos de teste.

Devido às características do conjunto de casos de teste utilizado, é possível obter um número máximo de discriminações equivalente ao número máximo de casos de teste analisados.

3.3.5.2 *Discrimination Rate* (Taxa de Discriminação)

A taxa de discriminação é calculada pela razão entre o número de discriminações pelo total de fraquezas a serem reportadas.

$$DiscriminationRate = \left(\frac{\#Discriminations}{\#Flaws} \right) \tag{3.5}$$

3.4 Análise Empírica do Comportamento de Técnicas de Análise

O NIST realiza, periodicamente, uma exposição em que reporta a análise de um conjunto de ferramentas de análise estática, denominado SATE (*Static Analysis Tool Exposition*). Ao final do processo, é gerado um documento, reportando os resultados obtidos, bem como o conjunto de dados utilizados para a obtenção desses resultados.

Os *reports* dados por cada ferramenta, durante a execução da mesma sobre o conjunto de casos de teste utilizado, são organizados de modo a se obter uma classificação comum entre as ferramentas. Assim, caso o reporte dado por uma determinada ferramenta, sobre a suíte de testes do Juliet, não esteja associado a uma CWE específica, essa associação é feita a partir do entendimento da descrição fornecida em cada *report*.

Após isso, as CWEs encontradas na suíte de testes do Juliet são combinadas e classificadas a partir de um grupo de CWEs. Esse agrupamento tem por objetivo diminuir a discrepância entre os reportes de cada ferramenta, associando-as segundo as características comuns de cada *report*.

3.4.1 SATE IV

Em Janeiro de 2013, foi publicado o relatório referente ao SATE IV, que teve seu início em 2011.

As etapas executadas no SATE IV estão listadas a seguir.

- Passo 1 - Preparação;
- Passo 2 - Fornecimento dos casos de teste via Web;
- Passo 3 - Execução das ferramentas sobre os casos de teste e relatório da execução;
- Passo 4 - Análise dos relatórios, por parte da organização;
- Passo 5 - Comparação entre os relaórios em um *workshop*;
- Passo 6 - Publicação dos resultados;

Os resultados obtidos para cada ferramenta são disponibilizados no formato XML[2]. Os *reports* dados por cada ferramenta possuem:

- ID - Um identificador único
- ID específico da ferramenta (Opcional)
- Um ou mais caminhos dos arquivos relacionados, possuindo:
  - ID (Opcional) - ID do caminho de arquivo, utilizado para diferenciar múltiplos caminhos relatados
  - Linha
  - Caminho
  - Framento de código (Opcional)
  - Explicação (Opcional)
- Nome da fraqueza (ex: *buffer overflow*)
- CWE ID

[2] http://www.w3.org/TR/REC-xml/

- Nível de severidade da *Weakness* (Dado pela ferramenta)
- Saída original dada pela ferramenta
- Avaliação do *report* feita por um ser humano (desconsiderado na análise da ferramenta)

É necessário extremo cuidado na intepretação dos dados disponibilizados pelo SATE IV. Não é objetivo dessa exposição a comparação de ferramentas. O uso de uma suíte de testes artificial e limitada tem como consequência o fato de que a ocorrência de cada vulnerabilidade, assim como as características apresentadas em cada subconjunto de CWEs não seja proporcional à encontrada em softwares reais. Nesse contexto, uma ferramenta que obtenha valores maiores do que os de outra ferramenta, não necessariamente mostrará esse comportamento em um contexto real.

Os *reports* dados por cada ferramenta são disponibilizados nessa formatação e de maneira mais sintética no *site* relativo ao SATE IV[3].

3.4.2 Ferramentas analisadas

Durante o SATE IV, foram analisadas algumas ferramentas, dentre as quais, as utilizadas na análise desse documento estão listadas na tabela (3.4.2) (em ordem alfabética).

Ferramenta	Versão	Linguagens
Cppcheck	1.49	C/C++
LDRA Testbed	8.5.3	C/C++
Monoidics INFER	1.5	C/C++
Parasoft C++test	9.1.1.25	C/C++
Red Lizard Software Goanna	9994 (devel. branch)	C/C++

Tabela 3 – Ferramentas Analisadas no SATE IV utilizadas nesse trabalho

Em uma breve análise sobre os resultados do SATE IV acerca das ferramentas citadas na tabela (3.4.2), verifica-se a eficiência dessas ferramentas a partir de todo o conjunto de CWEs passível de análise através do *Juliet* na versão 1.0. Entretanto, a partir análise granularizada de cada ferramenta na suíte de testes utilizada, é possível aferir resultados mais específicos que vão de encontro ao objetivo desse trabalho. Para isso, foram analisados a partir dos dados reportados e disponibilizados pelo SATE IV, a eficiência das ferramentas nas métricas listadas em subconjuntos da suíte de testes do Juliet.

[3] http://samate.nist.gov/SATE4.html

3.4.2.1 Cppcheck

O Cppcheck[4] é uma ferramenta de análise estática de código C/C++. É um software livre disponibilizado, atualmente, sob a licença GNU General Public License version 3.0 (GPLv3). Possui, como premissa, ser completamente livre de falsos positivos[5](MARJAMAKI, 2010).

A versão analisada no SATE IV possui uma série de análise de erros, categorizadas seguindo uma metodologia própria dos desenvolvedores da ferramenta. A funcionalidade da ferramenta que abrange *buffer overflow* é nomeada como *out of bounds checking* (Checagem de excedimento de limites, em tradução livre). Essa funcionalidade abrange, além da detecção de *buffer overflow*, a detecção de limites de fronteira, como a extrapolação do tamanho máximo permitido para um tipo de variável (uma variável do tipo inteiro, por exemplo), a atribuição de um valor negativo para uma variável não sinalizada (caracterizada, nas linguagens C e C++ com o especificado *unsigned*), a detecção de alocação de memória com tamanho negativo, entre outras.

A principal técnica de detecção citada pela documentação da ferramenta é a Análise do Fluxo de Dados (Seção 2.2.1.2.2. Ainda assim, essa análise é considerada simples pelos prósprios desenvolvedores(MARJAMAKI, 2010).

Foge ao escopo deste trabalho uma caracterização mais técnica das funcionalidades existentes na ferramenta. Entretanto, faz-se necessária uma análise sobre o comportamento do Cppcheck quanto ao modo como a análise estática é utilizada.

3.4.2.2 LDRA Testbed

É uma *engine* de análise estática e visualização de código. Esta *engine* tem por objetivo facilitar métricas de qualidades e normas de codificação, além de falhas a nível de código para o usuário, disponibilizando esses dados como insumo para a suíte de ferramentas LDRA.

Como *features* anunciadas pela ferramenta, destacam-se as listadas a seguir(LDRA, 2011).

- Análise estática do fluxo de dados: Rastreia as variáveis através do código-fonte e reporta qualquer uso anômalo.
- Análise do Fluxo de Informação: Analisa a inter-dependência de variáveis pelos caminhos de execução possíveis, através do código.
- Análise de laços: Relata a estrutura de laços e a profundidade de aninhamento dentro do código.

[4] http://cppcheck.sourceforge.net
[5] Sessão 3.2

- Análise de procedimentos recursivos: Executa as análises já listadas de maneira individual e em conjuntos de procedimentos recursivos (contextualizando cada anaĺise).

Caracteriza-se por ser uma ferramenta proprietária, impossibilitando uma análise mais profunda de seu funcionamento. Entretanto, sabe-se, por sua documentação, que utiliza análise de *Data Flow* e *Control Flow* para análise de código.

3.4.2.3 Monoidics INFER

O INFER é uma ferramenta de verificação automática voltada para a melhoria de segurança de memória em programas C. Ele tenta construir uma prova de composição do programa, através dos módulos que o constituem (funções e procedimentos)(CALCAGNO; DISTEFANO, 2011). *Bugs* são extraidos de falhas de tentativas de provas. Essa técnica, conhecida por lógica de separação, é caracterizada por (O'HEARN, 2012).

Dentre as funcionalidades da ferramenta, destaca-se neste trabalho a *deep-heap analysis* (também conhecida como *shape analysis*) na presença de alocação dinâmica de memória. Segundo (CALCAGNO; DISTEFANO, 2011), o domínio abstrato da ferramenta pode interpretar, de forma precisa, complexas estruturas de dados alocadas dinamicamente, bem como listas isoladas que referenciem apenas entre si, sejam circulares, agrupadas com outras listas, entre outros.

3.4.2.4 Parasoft C++test

A Parasoft C++test® é uma ferramenta de testes de desenvolvimento integrado, que tem como bandeira a automação de melhores práticas de desenvolvimento, visando melhorar a produtividade da equipe de desenvolvimento de software e a qualidade de software(PARASOFT, 2013).

É um software proprietário, com suporte às plataformas Windows, Linux e Solaris UltraSPARC.

A análise estática feita por essa ferramenta se baseia na Análise do fluxo de dados, utilizando *flow based static analysis* e *pattern-based static analysis*.

Em relação à *pattern-based static analysis*, a documentação da ferramenta afirma que a Parasoft utiliza um conjunto de milhares de regras para encontrar padrões de código que levem a problemas de confiabilidade, desempenho e segurança.

Dadas essas características, a análise estática baseada em padrões, implementada pela ferramenta, difere da análise com Detecção de Padrões (Seção 2.2.1.2.1).

3.4.2.5 Red Lizard Software Goanna

A *Goanna Static Analysis Tool*, é uma ferramenta de análise estática voltada, segundo(FEATURES..., 2009), para a detecção de falhas de difícil localização em códigos C/C++. Dentre as falhas que essa ferramenta procura encontrar estão violação de acesso, invasão de memória em strings e arrays, vazamento de memória, falhas de segurança e erros aritméticos.

É um software proprietário, pertencente à Red Lizard. Este fator impossibilita um conhecimento mais detalhado sobre os métodos de análise utilizados pela ferramenta. Entretanto, a documentação oficial (FEATURES..., 2009) caracteriza algumas técnicas para detecção de vulnerabilidades, por parte da Goanna. Essas técnicas estão listadas a seguir.

- Model Checking (Seção 2.2.1.2.6)
  - A ferramenta constrói uma representação do fluxo de cada função C/C++ com informações de interesse. A estrutura construída é então analisada a partir de uma checagem de modelos, permitindo identificar comportamentos não esperados.
- Análise Interprocedural
  - Essa funcionalidade, dita como patenteada pela desenvolvedora da ferramenta, utiliza uma análise do fluxo de dados e das dependências desses dados em todo o programa. A análise desse fluxo permite, por exemplo, a detecção do uso de um ponteiro nulo em um determinado ponto do fluxo do programa.
  - É interpretada, no contexto desse trabalho, como uma variação da Análise do Fluxo de Dados (Seção 2.2.1.2.2)
- Rastreamento de dados abstratos
  - A ferramenta busca analisar o comportamento da variação de valores de variáveis ao longo do fluxo do programa. Assim, determinando os valores mínimos e máximos de uma variável que determine o índice de acesso a um determinado *buffer*, por exemplo, é possível inferir a ocorrência de um *buffer overflow*.

3.4.3 Classificação de *buffer overflows* a partir de seus atributos

Foram analisados alguns atributos que caracterizam um *buffer overflow*, como listado na tabela (2.3.1). Destes, foram selecionados os atributos:

- Tipo de acesso;

  - Leitura;
  - Escrita;

- Limite do *Buffer*;
  - Inicial;
  - Final;

- Local da Memória;
  - *Heap*;
  - *Stack*;

Para cada variação de atributo, foram selecionados os conjuntos de casos de teste correspondentes, de acordo com a CWE relacionada:

- Leitura - CWEs 126 e 127;
- Escrita - CWE 124;
- Limite Inicial - CWEs 124 e 127;
- Limite Final - CWE 126;
- Local da Memória: Heap - CWE 121;
- Local da Memória: Stack - CWE 122;

3.5 Ferramenta de Análise

Foi desenvolvida uma ferramenta em linguagem Perl para análise dos resultados obtidos no SATE IV. Uma das principais características da linguagem utilizada é a eficiência no tratamento de dados em linguagem natural (texto puro) (PRECHELT, 2000). Assim, um documento de *report*, de cerca de 1,3 milhão de linhas, é analisado e seus dados utilizados no cálculo das métricas de um subconjunto de CWEs em cerca de 16 segundos, por parte da ferramenta. O *report* do conjunto de casos de teste disponibilizados no Juliet seguem o formato exemplificado no Anexo B.

A figura (2) mostra um diagrama de sequências, exemplificando o comportamento da ferramenta para calcular as métricas descritas em **Métricas** (Seção 3.3).

Os resultados obtidos e métricas calculadas são disponibilizados sob a forma de tabela (*array* de *array*, na linguagem Perl). Uma classe de Output, existente na ferramenta, possibilita a disponibilização da tabela no formato LaTeXou CSV (até o momento).

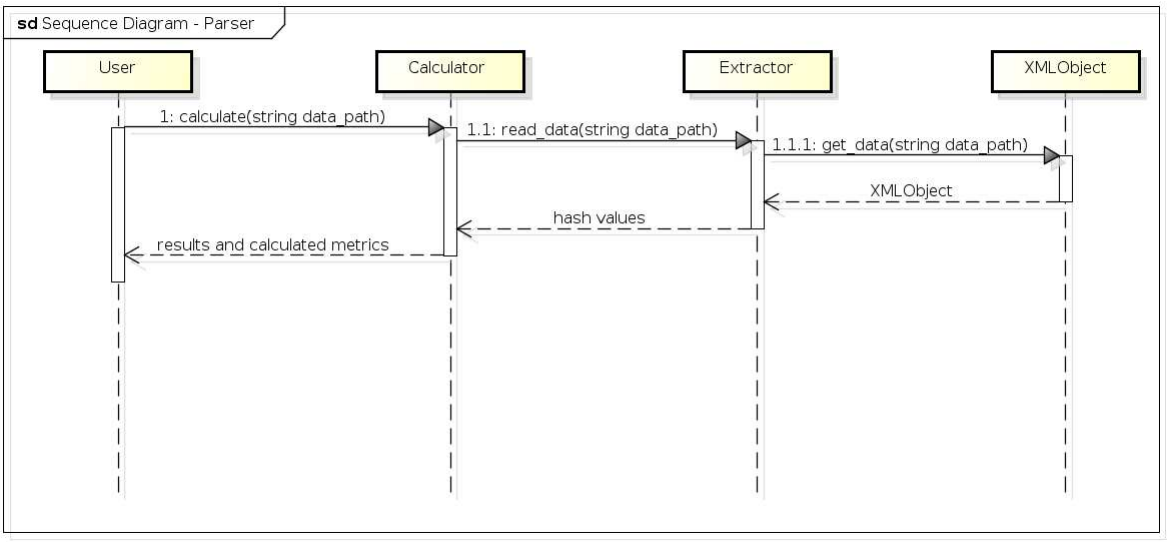

Figura 2 – Diagrama de Sequência da *Sate Report Analysis Tool*

As tabelas, obtidas em Resultados e Métricas Obtidas (Seção 4.1), foram geradas no formato LaTeX, para esse documento.

A ferramenta tem funcionamento simples e dispõe, até o momento, de duas funcionalidades principais: o cálculo das métricas das ferramentas para um dado subgrupo de CWEs (os dois são informados como parâmetro da funcionalidade) e o cálculo conjunto de duas ferramentas em um dado subgrupo de CWEs (no caso dessa funcionalidade, além das CWEs desejadas, são informadas duas ferramentas para serem analisadas conjuntamente).

A partir das métricas obtidas, foram gerados gráficos comparativos para cada subconjunto de CWEs analisado. Os gráficos foram desenvolvidos utilizando o Gnuplot[6].

Tanto o código-fonte do *parser* desenvolvido, quanto os scripts para geração dos gráficos utilizados, estão disponíveis em https://github.com/alexandreab/satereportanalysis[7]. O Apêndice A descreve os principais módulos que compõem esse a ferramenta.

6 http://www.gnuplot.info/
7 Último acesso em 6 de Fevereiro de 2015

4 Resultados

A partir dos subconjuntos de CWEs selecionados, foi feita uma leitura automatizada (*parse*) dos resultados referentes a cada subconjunto e calculados o total de Verdadeiros positivos, falsos positivos, falsos negativos e indefinidos (aqui caracterizados como verdadeiros negativos).

Os **Resultados e Métricas Obtidas** (Seção 4.1) foram agrupados por subgrupos de CWEs. Na **Análise dos resultados obtidos** (Seção 4.2), esses resultados são disponibilizados sob a forma de gráficos, e analisados por ferramenta, comparando-se as métricas obtidas segundo a taxonomia do *buffer overflow*.

4.1 Resultados e Métricas Obtidas

Nas subseções a seguir são evidenciadas os resultados obtidos e as métricas calculadas para cada subconjunto.

4.1.1 Resultados obtidos para cada subconjunto de CWEs analisado

Estão listados aqui os resultados agrupados por característica, segundo os Subgrupos de *Buffer Overflow* (Seção 2.3.1.1).

4.1.1.1 Tipo de acesso

Conforme já descrito, o acesso a um determinado espaço de memória pode ser feito como escrita ou como leitura. A partir dessas características, as seções seguintes descrevem os resultados obtidos nos grupos de caso de teste de *buffer overflow* em acesso de leitura (formado pelas CWEs 126 e 127) e em acesso de escrita (formado pela CWE 124).

4.1.1.1.1 Leitura - CWEs 126 e 127

Ferramenta/Valor	FN	FP	TN	TP
cppcheck	2352	0	2352	0
ldra	2352	0	2352	0
monoidics	2268	0	2352	84
parasoft	1644	0	2352	708
redlizard	2328	204	2148	24

Tabela 4 – Buffer Overflows relacionados a Leitura de Buffer - 2352 Casos de teste

A tabela (4) mostra os *reports* obtidos no subconjunto de CWEs relacionadas à leitura de *buffer*. Nesse subconjunto, as ferramentas Cppcheck e Ldra não reportaram nenhum positivo (verdadeiro ou falso). Este fato impossibilita o cálculo de algumas métricas calculadas nesse trabalho. Além disso, apenas a ferramenta Redlizard reportou falsos positivos.

4.1.1.1.2 Escrita - CWE 124

Ferramenta/Valor	FN	FP	TN	TP
cppcheck	1470	0	1470	0
ldra	1470	0	1470	0
monoidics	1386	0	1470	84
parasoft	1176	0	1470	294
redlizard	1470	0	1470	0

Tabela 5 – Buffer Overflows relacionados a Escrita de Buffer - 1470 Casos de teste

A tabela (5) mostra os *reports* obtidos no subconjunto de CWEs relacionadas à escrita de *buffer*. Nesse subconjunto, além das ferramentas Cppcheck e Ldra, a Relizard também não foi capaz de reportar nenhum positivo (verdadeiro ou falso). Nenhuma ferramenta reportou Falso positivo.

4.1.1.2 Limite de acesso

As seções seguintes descrevem os resultados obtidos nos grupos de caso de teste de *buffer overflow* que ocorrem quando o acesso ultrapassa o limite inicial do *buffer* (formado pelas CWEs 124 e 127) e quando ultrapassa o limite final (formado pela CWE 126).

4.1.1.2.1 Limite Inicial - CWEs 124 e 127

Ferramenta/Valor	FN	FP	TN	TP
cppcheck	2940	0	2940	0
ldra	2940	0	2940	0
monoidics	2772	0	2940	168
parasoft	2205	0	2940	735
redlizard	2940	0	2940	0

Tabela 6 – Buffer Overflows relacionados ao Limite Inicial do Buffer - 2940 Casos de teste

A tabela (6) mostra os *reports* obtidos no subconjunto de CWEs relacionadas ao acesso na área inicial do *buffer*. Nesse subconjunto, novamente, apenas as ferramentas Monoidics Infer e Parasoft reportaram verdadeiros positivos.

4.1.1.2.2 Limite Final - CWE 126

Ferramenta/Valor	FN	FP	TN	TP
cppcheck	882	0	882	0
ldra	882	0	882	0
monoidics	882	0	882	0
parasoft	615	0	882	267
redlizard	858	204	678	24

Tabela 7 – Buffer Overflows relacionados ao Limite Final do Buffer - 882 Casos de teste

A tabela (7) mostra os *reports* obtidos no subconjunto de CWEs relacionadas ao acesso na área inicial do *buffer*. Nesse subconjunto, novamente, apenas as ferramentas Monoidics Infer e Parasoft reportaram verdadeiros positivos.

4.1.1.3 Local da Memória

As seções seguintes descrevem os resultados obtidos nos grupos de caso de teste de *buffer overflow* que ocorrem *buffer* alocado em *Heap* (formado pela CWE 121) ou *Stack* (formado pela CWE 122).

4.1.1.3.1 Heap - CWE 121

Ferramenta/Valor	FN	FP	TN	TP
cppcheck	2930	0	2946	16
ldra	2935	6	2940	11
monoidics	2671	44	2902	275
parasoft	2106	0	2946	840
redlizard	2610	314	2632	336

Tabela 8 – Buffer Overflows relacionados a Buffers em Heap - 2946 Casos de teste

A tabela (8) mostra os *reports* obtidos no subconjunto de CWEs relacionadas ao acesso de *buffers* armazenados em Heap. Ao contrário dos outros subconjuntos, todas as ferramentas reportaram verdadeiros positivos nesse grupo. Entretanto, apenas as ferramentas Cppcheck e Parasoft não retornaram falsos positivos.

4.1.1.3.2 Stack - CWE 122

A tabela (9) mostra os *reports* obtidos no subconjunto de CWEs relacionadas ao acesso de *buffers* armazenados em Stack. Neste subconjunto, apenas a ferramenta Ldra não reportou nenhum positivo (verdadeiro ou falso). Com exceção desta, apena a Parasoft não retornou nenhum falso positivo.

Ferramenta/Valor	FN	FP	TN	TP
cppcheck	3171	24	3209	62
ldra	3233	0	3233	0
monoidics	2975	22	3211	258
parasoft	2393	0	3233	840
redlizard	3079	154	3079	154

Tabela 9 – Buffer Overflows relacionados a Buffers em Stack - 3233 Casos de teste

4.1.2 Métricas obtidas para cada subconjunto de CWEs analisado

A partir dos resultados caracterizados na Seção anterior, foram calculadas algumas métricas baseadas em ((CAS, 2011)). Algumas dessas métricas não puderam ser calculadas por conter fatores inválidos, que impossibilitam o cálculo desejado. Os valores não calculados estão representados por **NULL**.

A análise desses resultados está caracterizada na Seção 4.2 (Análise dos resultados).

4.1.2.1 Tipo de acesso

4.1.2.1.1 Leitura - CWEs 126 e 127

Ferramenta/Valor	Discrim	Discrim Rate	F-Score	FP Rate	Precision	Recall
cppcheck	0	0	NULL	NULL	NULL	0
ldra	0	0	NULL	NULL	NULL	0
monoidics	84	0.03571	0.06897	0	1	0.03571
parasoft	708	0.30102	0.46275	0	1	0.30102
redlizard	12	0.0051	0.0186	0.89474	0.10526	0.0102

Tabela 10 – Buffer Overflows relacionados a Leitura de Buffer - Metricas

Os valores "NULL"exibidos na tabela (10) referem-se às métricas das ferramentas Cppcheck e Ldra que não puderam ser calculadas, por insuficiência de dados (verdadeiros positivos, mais especificamente).

4.1.2.1.2 Escrita - CWE 124

Valores "NULL"também são exibidos na tabela (11) e, além das ferramentas Cppcheck e Ldra, também estão presentes nos resultados da ferramenta Redlizard que, como as duas primeiras, não reportou Verdadeiros positivos que permitissem o cálculo dessas métricas.

Ferramenta/Valor	Discrim	Discrim Rate	F-Score	FP Rate	Precision	Recall
cppcheck	0	0	NULL	NULL	NULL	0
ldra	0	0	NULL	NULL	NULL	0
monoidics	84	0.05714	0.10811	0	1	0.05714
parasoft	294	0.2	0.33333	0	1	0.2
redlizard	0	0	NULL	NULL	NULL	0

Tabela 11 – Buffer Overflows relacionados a Escrita de Buffer - Metricas

4.1.2.2 Limite de acesso

As seções seguintes disponibilizam as métricas calculadas com base nos subgrupos classificados de acordo com a área de memória acessada (limite inicial e limite final).

4.1.2.2.1 Limite Inicial - CWEs 124 e 127

Ferramenta/Valor	Discrim	Discrim Rate	F-Score	FP Rate	Precision	Recall
cppcheck	0	0	NULL	NULL	NULL	0
ldra	0	0	NULL	NULL	NULL	0
monoidics	168	0.05714	0.10811	0	1	0.05714
parasoft	735	0.25	0.4	0	1	0.25
redlizard	0	0	NULL	NULL	NULL	0

Tabela 12 – Buffer Overflows relacionados ao Limite Inicial do Buffer - Metricas

4.1.2.2.2 Limite Final - CWE 126

Ferramenta/Valor	Discrim	Discrim Rate	F-Score	FP Rate	Precision	Recall
cppcheck	0	0	NULL	NULL	NULL	0
ldra	0	0	NULL	NULL	NULL	0
monoidics	0	0	NULL	NULL	NULL	0
parasoft	267	0.30272	0.46475	0	1	0.30272
redlizard	12	0.01361	0.04324	0.89474	0.10526	0.02721

Tabela 13 – Buffer Overflows relacionados ao Limite Final do Buffer - Metricas

4.1.2.3 Local da Memória

As duas seções seguintes disponibilizam as métricas calculadas com base nos subgrupos classificados de acordo com o modo como os *buffers* de memória presentes nos casos de teste são alocados. Essa alocação pode ser feita em *Heap* ou *Stack*.

Dentre os subgrupos de CWEs analisados, os relativos à CWE 121 e à CWE 122 foram os que reportaram mais ocorrências de verdadeiro positivo, possibilitando o cálculo das métricas da maioria das ferramentas analisadas.

4.1.2.3.1 Heap - CWE 121

Ferramenta/Valor	Discrim	Discrim Rate	F-Score	FP Rate	Precision	Recall
cppcheck	16	0.00543	0.0108	0	1	0.00543
ldra	6	0.00204	0.00742	0.35294	0.64706	0.00373
monoidics	231	0.07841	0.16845	0.13793	0.86207	0.09335
parasoft	840	0.28513	0.44374	0	1	0.28513
redlizard	22	0.00747	0.18687	0.48308	0.51692	0.11405

Tabela 14 – Buffer Overflows relacionados a Buffers em Heap - Metricas

A tabela (14) mostra as métricas obtidas no conjunto de testes relacionados à CWE 121. É interessante observar que, nesse conjunto de casos de teste, todas as métricas, de todas as ferramentas foram obtidas. Isso ocorreu porque todas as ferramentas reportaram verdadeiros positivos em pelo menos um caso de teste.

4.1.2.3.2 Stack - CWE 122

Ferramenta/Valor	Discrim	Discrim Rate	F-Score	FP Rate	Precision	Recall
cppcheck	38	0.01175	0.03736	0.27907	0.72093	0.01918
ldra	0	0	NULL	NULL	NULL	0
monoidics	236	0.073	0.14688	0.07857	0.92143	0.0798
parasoft	840	0.25982	0.41247	0	1	0.25982
redlizard	0	0	0.08698	0.5	0.5	0.04763

Tabela 15 – Buffer Overflows relacionados a Buffers em Stack - Metricas

Na tabela (15), observa-se que apenas a ferramenta Ldra não reportou verdadeiros positivos. Isso impossibilitou os cálculos das métricas analisadas para a ferramenta em questão.

4.2 Análise dos resultados obtidos

4.2.1 Análise das métricas obtidas por ferramenta

As subseções à seguir mostram, graficamente, as métricas obtidas da análise de das ferramentas em cada subgrupo de *buffer overflow*. As ferramentas **Cppcheck** e **Ldra** tiveram métricas apenas no subgrupo relacionado ao tipo de alocação do *buffer* de memória. E a **Ldra** teve métricas apenas em *buffer overflows* alocados em *Heap*.

4.2.1.1 Cppcheck

A figura 3 compara as métricas obtidas da ferramenta Cppcheck de acordo com o tipo de alocação de memória.

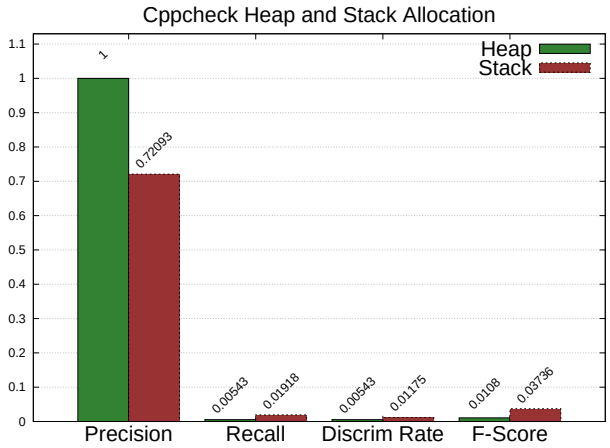

Figura 3 – Métricas obtidas do report da ferramenta Cppcheck - Heap e Stack

Ressalta-se, aqui, uma maior precisão do Cppcheck em reportar *buffer overflows* do tipo *heap*. Em *buffers* de memória do tipo *stack*, a ferramenta reportou falsos positivos, o que diminuiu a precisão calculada. Já os valores de *Recall* e *F-Score* são extremamente baixos e próximos, dificultando uma análise independente de margem de erro (por ser uma análise numérica do comportamento da ferramenta e, portanto, com uma margem de erro). Relevando-se esses fatores, a ferramenta evidenciou métricas maiores para *buffer overflows* ocorridos em *stack*. Esse comportamento se repete quando se analisa o *discrimination rate*, caracterizando uma melhor diferenciação, por parte da ferramenta, entre verdadeiros positivos e falsos positivos nos casos de teste relacionados à ocorrência de *buffer overflow* alocados em *stack*.

Assim, a partir das técnicas de análise estáticas que a ferramenta Cppcheck implementa, dentre as quais uma implementação básica da análise de fluxo de dados, obteve-se uma maior precisão em detectar *buffer overflows* alocados em *Heap*, mas um maior *Recall* e *Discrim Rate* nos casos de teste de *buffer overflow* alocado em *Stack*.

4.2.1.2 Ldra

A figura 4 compara, graficamente, as métricas obtidas pela ferramenta Ldra, para *Heap* e *Stack*.

Em relação a *buffer overflows* ocorridos em *Heap*, a ferramenta possuiu uma precisão de 64,7%. Entretanto, é preciso cautela para os objetivos dessa métrica e para o modo como o cálculo desta é feito. Esse valor foi obtido porque a ferramenta reportou 17 ocorrências de *buffer overflow* e, destas, 6 foram falsos positivos. Analisando o *Recall*, e

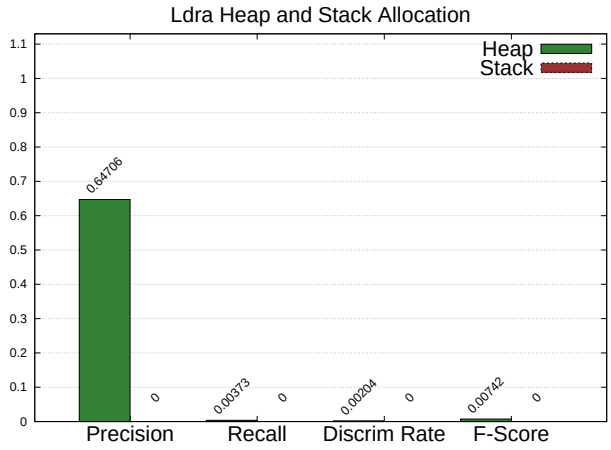

Figura 4 – Métricas obtidas do report da ferramenta Ldra - Heap e Stack

até mesmo o *F-Score*, que é obtida através da precisão e do *Recall*, percebe-se que houve uma taxa muito baixa de report de ocorrências (há um total de 2946 casos de teste voltados para esse subgrupo analisado). Assim, as métricas obtidas devem ser interpretadas com cautela.

Em relação aos *buffer overflows* ocorridos em *Stack*, a ferramenta não reportou nenhuma ocorrência da vulnerabilidade em questão. Desse modo, o cálculo das métricas não pôde ser feito e a comparação do comportamento da ferramenta em cada característica desse subgrupo não ocorreu.

Assim, o modo como a análise de fluxo de dados implementada pela ferramenta não foi capaz de detectar *buffer overflows* ocorridos em *Stack*

4.2.1.3 Parasoft

A ferramenta Parasoft obteve resultados em todos os subgrupos definidos. Os gráficos a seguir comparam as métricas obtidas para cada subgrupo analisado. Para todos os subgrupos analisados, A ferramenta não reportou nenhum Falso Positivo, fazendo com que os valores de *Recall* e *Discrim Rate* sejam iguais.

A análise das métricas obtidas relacionadas a leitura e escrita (exibidas na figura 5), evidenciam uma maior eficiência na identificação de *buffer overflows* ocorridos em acesso de leitura do que de escrita. Ainda que a precisão apresentada seja a mesma para ambos os casos, A discrepância entre os valores das outras métricas é consideravelmente alto.

Já no que se refere aos limites do *buffer* onde a vulnerabilidade pode ocorrer, a **Parasoft** obteve valores mais altos em *buffer overflows* ocorridos no limite final do *buffer*, como mostra a firgura 6. Vale ressaltar que a diferença nos resultados dos dois tipos de limite é bem menor que os resultados relacionados ao tipo de acesso.

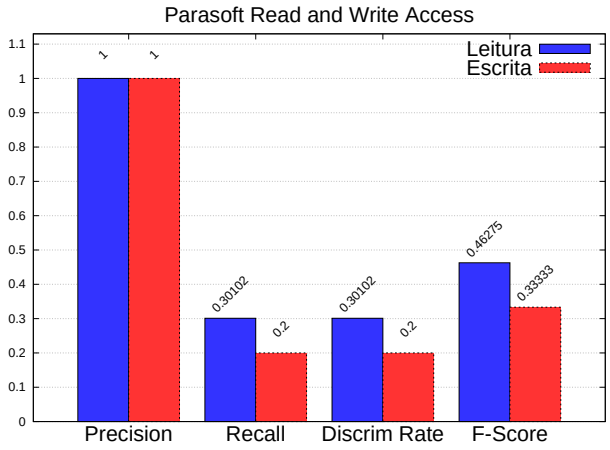

Figura 5 – Métricas obtidas do report da ferramenta Parasoft - Leitura e Escrita

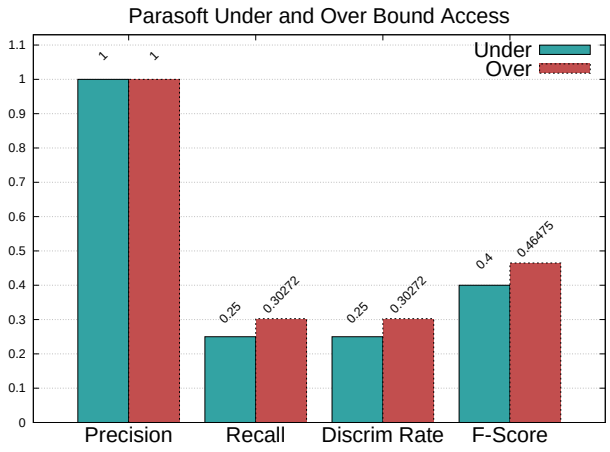

Figura 6 – Métricas obtidas do report da ferramenta Parasoft - Limite Inicial e Final

Do mesmo modo, em relação ao tipo de alocação de memória do *buffer*, a ferramenta mostrou uma eficiência maior em *buffers* alocados em *Heap* (figura 7). Assim como na situação anterior, a diferança na identificação de vulnerabilidades para os dois tipos de alocação de *buffer* foi pequena.

4.2.1.4 Monoidics

A ferramenta **Monoidics** apresentou diferentes características para cada subgrupo analisado. Das seis variações possíveis para os três atributos do *buffer overflow* analisados, a ferramenta não reportou verdadeiros positivos para as vulnerabilidades ocorridas no limite posterior do *buffer*. A figura 8 compara as métricas obtidas em Leitura e Escrita. Para ambos os casos, a ferramenta apresentou uma boa precisão. Entretanto, os valores obtidos para as outras métricas foram consideravelmente baixos, com uma maior vantagem para os valores relacionados à Escrita do *buffer*.

A figura 9 mostra que a ferramenta não foi capaz de reportar ocorrências de vul-

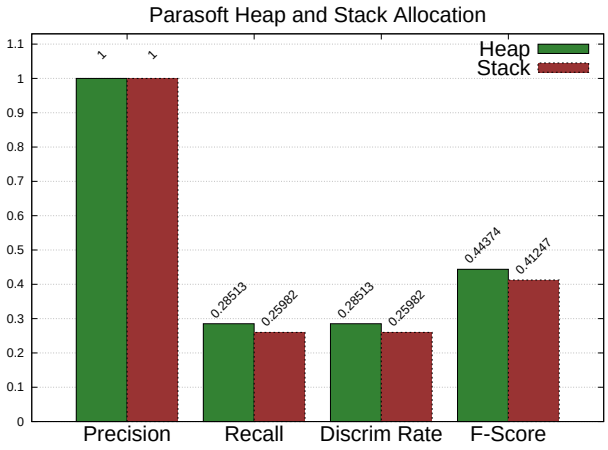

Figura 7 – Métricas obtidas do report da ferramenta Parasoft - Heap e Stack

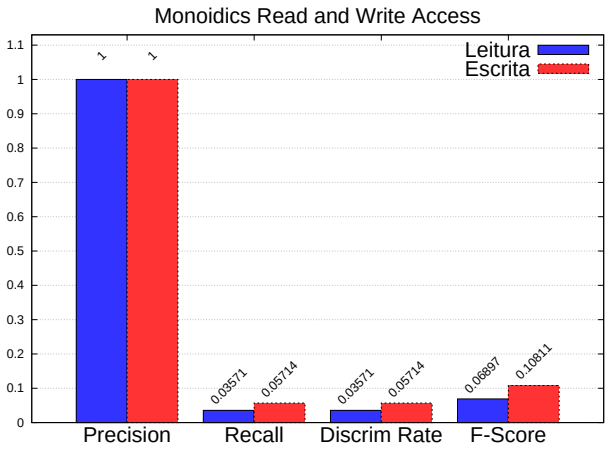

Figura 8 – Métricas obtidas do report da ferramenta Monoidics - Leitura e Escrita

nerabilidades no limite final do *buffer*. Apenas vulnerabilidades no limite inicial foram reportadas.

Quanto à comparação entre *buffer overflows* ocorridos em *buffers* alocados em *heap* e *stack*, a figura 10 mostra que a **Monoidics Infer** reportou falsos positivos para ambos os casos, mas que obteve uma maior precisão em *buffer overflows* ocorridos em *stack*. Entretanto essa diferença se inverte para as outras métricas, evidenciando que a ferramenta foi mais eficiente, na suíte de testes analisada, em reportar *buffer overflows* ocorridos em *heap*.

A diferença das métricas obtidas para cada tipo de alocação foi consideravelmente pequena, sendo menor ainda na métrica *Discrim Rate* (que mostra a capacidade da ferramenta de discernir entre um falso positivo e um verdadeiro positivo, sobre o total de vulnerabilidades, do tipo, existentes).

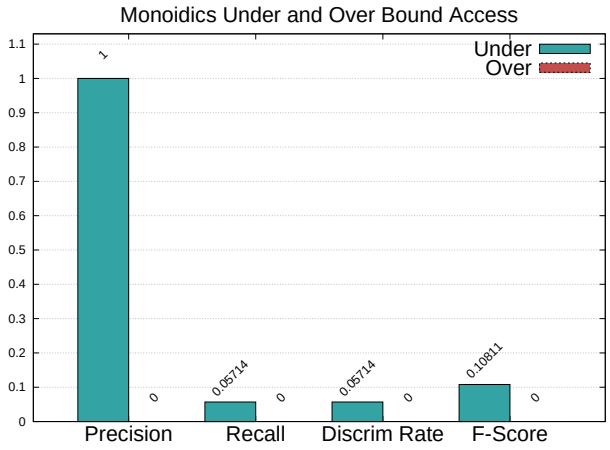

Figura 9 – Métricas obtidas do report da ferramenta Monoidics - Limite Inicial e Final

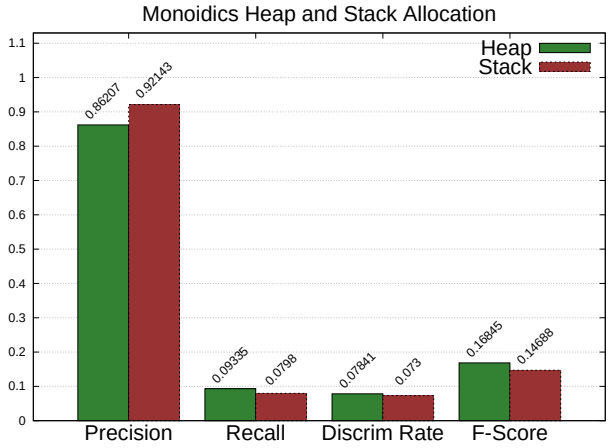

Figura 10 – Métricas obtidas do report da ferramenta Monoidics - Heap e Stack

4.2.1.5 RedLizard

Dentre os subgrupos analisados A ferramenta **RedLizard** não foi capaz de reportar *buffer overflows* ocorridos em escrita de *buffer* ou no limite final dos mesmos.

A figura 11 mostra as métricas obtidas nos casos de teste em que ocorrem *buffer overflows* em leitura de *buffer*. As métricas dos casos de teste em que a vulnerabildiade ocorre em escrita de *buffer* não puderem ser calculadas.

Os valores mostrados evidenciam que a ferramenta reportou um valor muito alto de falsos positivos em relação ao total de positivos existentes. Somados a isso, as outras métricas evidenciam um índice baixo de ocorrências reportadas no na suíte de testes utilizada.

A figura 12 possui características semelhantes ao caso anterior. Nesse contexto, a ferramenta não foi capaz de identificar *buffer overflows* ocorridos no limite inicial do *buffer*. Já as vulnerabilidades do tipo ocorridas no limite final do buffer foram reportadas com uma precisão baixa, já que a ferramenta reportou muitos falsos positivos.

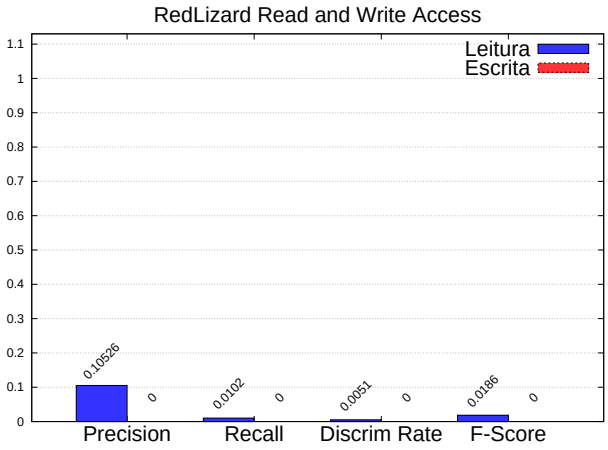

Figura 11 – Métricas obtidas do report da ferramenta RedLizard - Leitura e Escrita

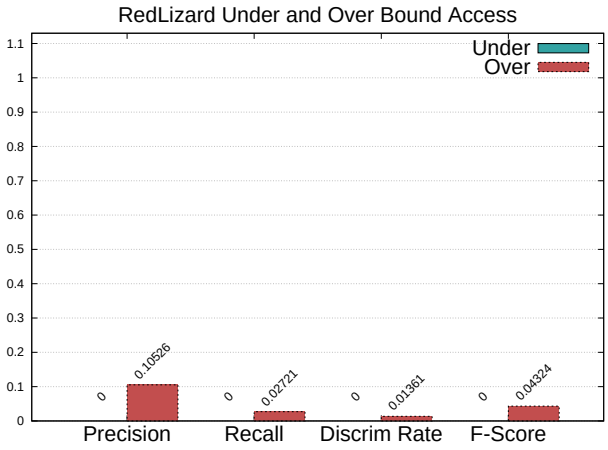

Figura 12 – Métricas obtidas do report da ferramenta RedLizard - Limite Inicial e Final

Dentre os subgrupos analisados, a comparação entre as vulnerabilidades ocorridas de acordo com o tipo de alocação foram as únicas viáveis, visto que, tanto para *heap* quanto para *stack*, a ferramenta reportou valores que possibilitassem o cálculo das métricas. Essas métricas são exibidas no gráfico da figura 13. Pelo gráfico é possível perceber uma maior eficiência da ferramenta para os *buffer overflows* ocorridos em *heap*. Tanto a precisão da ferramenta, quanto a capacidade de reportar verdadeiros positivos e discerni-los de falsos positivos foram superiores nos casos de teste voltados para *buffer overflows* ocorridos em *heap* nos casos de teste voltados para *stack buffer overflows*.

As subseções à seguir mostram, graficamente, as métricas obtidas da análise de das ferramentas em cada subgrupo de *buffer overflow*. As ferramentas **Cppcheck** e **Ldra** tiveram métricas apenas no subgrupo relacionado ao tipo de alocação do *buffer* de memória. E a **Ldra** teve métricas apenas em *buffer overflows* alocados em *Heap*.

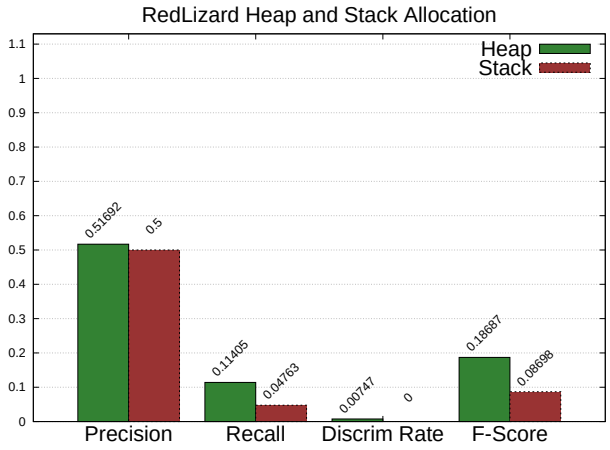

Figura 13 – Métricas obtidas do report da ferramenta RedLizard - Heap e Stack

4.2.2 Análise das métricas obtidas por técnica de análise

4.2.2.1 Análise de fluxo de dados

A análise de fluxo de dados é implementada, em diferentes níveis, pelas ferramentas **Cppcheck, LDRA Testbed, Red Lizard Software Goanna** e **Parasoft C++test**.

4.2.2.1.1 Tipo de acesso

No que se refere ao tipo de acesso, as ferramentas Cppcheck e LDRA não reportaram verdadeiros positivos para os casos de teste de leitura de *buffer* que continham a vulnerabilidade em questão. Já para os acessos de escrita de buffer, além das ferramentas Cppcheck e LDRA, a Red Lizard também não reportou verdadeiro positivo.

Comparando-se os tipos de acesso analisados, as ferramentas **Parasoft** e **Red Lizard**, que implementam a analíse de fluxo de dados, demonstraram uma maior eficiência na detecção de *buffer overflows* ocorridos em acessos de leitura do que nos acessos de escrita.

Ainda que os níveis de implementação da técnica por cada ferramenta sejam diferentes, é perceptível, no contexto das ferramentas analisadas e na suíte de testes utilizada, que essa técnica de análise não é o fator determinante para detecção de *buffer overflow*, visto que o desempenho das ferramentas, como um todo, foi baixo.

4.2.2.1.2 Local de acesso

No que se refere ao local de acesso onde ocorre o *buffer overflow*, apenas a ferramenta **Parasoft**, dentre as que implementam a análise de fluxo de dados, reportou verdadeiros positivos para os casos de teste relacionados a *buffer overflow* no início do

buffer. Quanto aos casos de teste relacionados ao limite final do *buffer*, além da **Parasoft**, a **Red Lizard** também reportou resultados.

As duas ferramentas que reportaram ocorrências de *buffer overflow* nesse subgrupo analisado demonstram uma maior eficiência na detecção da vulnerabilidade quando a mesma ocorreu no limite final do *buffer*.

A ferramenta **Parasoft** apresentou uma diferença de cerca de 5% para as métricas de *Recall* e *Discrim Rate* e de cerca de 6% para a métrica *F-Score*. Diferença esta, favorável a ocorrências da vulnerabilidade no final do *buffer*.

A ferramenta **Red Lizard** apresentou, para os casos de estouro de *buffer* ocorridos no final do mesmo, taxas de *Recall* e *Discrim Rate* inferiores a 3%, além de uma precisão de apenas 10%. Assim, a taxa de *F-Score* ficou em cerca de 4,3%. Em contra-partida, nada foi reportado para a *Buffer Underread* ou *Buffer Underwrite*.

Visto que o primeiro grupo analisado (*Buffer Overread*) possui apenas acessos de leitura, é preciso considerar a possibilidade de que a ferramenta tenha reportado ocorrências da vulnerabilidade por esta ocorrer em acesso de leitura (essa característica da *Red Lizard* foi observada na Seção anterior). Entretanto, sob essa mesma perspectiva, é possível observar que o subgrupo de *buffer overflows* ocorridos no limite inferior do *buffer* é composto pelas CWEs 124 e 127 (***Buffer Underwrite*** e ***Buffer Underread***, respectivamente). Dos 2352 casos de teste pertencentes a esse subgrupo, 2140 são relativos à CWE 127, que se caracteriza pelo acesso de leitura no início do *buffer*. Mesmo assim, a ferramenta não foi capaz de reportar nenhuma ocorrência da vulnerabilidade nos casos de teste relacionados a essa CWE, evidenciando que não foi o tipo de acesso que definiu os resultados obtidos.

4.2.2.1.3 Tipo de alocação

Quanto aos grupos de caso de teste relacionados ao tipo de alocação de memória, todas as ferramentas que implementam a análise do fluxo de dados reportaram ocorrências de *buffer overflow* em *Heap*. Já nos casos de teste em que a vulnerabilidade ocorre em *Stack*, apenas a **LDRA Testbed** não foi capaz de identificar essas ocorrências.

Em todas as ferramentas que implementam a análise de fluxo de dados, a precisão calculada foi maior para *buffer overflows* ocorridos em *Heap*. A **Parasoft** apresentou precisão 1 para ambos os tipos de alocação. No que se refere às outras métricas, apenas a **Cppcheck** apresentou melhores resultados para *buffer overflows* ocorridos em *Stack*, ainda que também sejam valores muito baixos. Todas as outras apresentaram melhores resultados em ocorrências em *Heap*.

Como a implementação dessa técnica de análise na ferramenta **Cppcheck** é considerada simples, pode-se inferir que a análise de fluxo de dados apresentou melhores

resultados para identificar *buffer overflows* ocorridos em *Heap*.

4.2.2.2 Lógica de Separação e *Shape Analysis*

A lógica de separação e a *Shape Analysis* são técnicas de análise estática implementadas pela ferramenta **Monoidics INFER**.

4.2.2.2.1 Tipo de acesso

Em relação ao tipo de acesso, a ferramenta que implementa essas duas técnicas se mostrou mais eficiente na detecção de *buffer oveflows* ocorridos na escrita de *buffer*. Esse resultado constrasta com o obtido para análise de fluxo de dados, nesse mesmo contexto.

Ressalta-se que a diferença entre as métricas de cada subgrupo foi de cerca de 2%, para *Recall* e *Discrim Rate* e de cerca de 3% para *F-Score*. Ainda que tenham sido avaliados 2532 casos de teste com acesso de leitura, e 1470 casos de teste com acesso de escrita, os valores obtidos são relativamente próximos e este fator deve ser considerado em contextos diferentes da análise empírica apresentada nesse trabalho.

4.2.2.2.2 Local de acesso

Em relação aos casos de teste voltados à análise do local de acesso, foram reportados *buffer overflows* ocorridos somente no limite inferior do *buffer*. Embora as vulnerabilidades não reportadas pela ferramenta relativas ao acesso no final do *buffer* sejam relativos somente à CWE 126 (*Buffer Overread*), que possui exclusivamente casos de teste com acessos de leitura, observa-se que, a partir do tipo de acesso descrito na Seção anterior, que a ferramenta é capaz de reportar vulnerabilidades ocorridas em acesso de leitura.

Assim, infere-se, dentro do contexto desse trabalho, que as técnicas de lógica de separação e *shape analysis* possuem, juntas, maior eficiência na detecção de *buffer overflows* quando estes ocorrem no limite inicial do *buffer*.

4.2.2.2.3 Tipo de alocação

Assim como a análise de fluxo de dados, as técnicas de Lógica de Separação e *Shape Analysis* juntas, possuem uma maior eficiência na detecção de *buffer overflows* quando estes ocorrem em *buffers* alocados em *Heap*. Entretanto, a diferença entre as métricas obtidas por esses dois tipos de acesso é menor do que as diferenças encontradas, de modo geral, nas ferramentas que implementam a análise de fluxo de dados.

4.2.2.3 Checagem de Modelos

A checagem de modelos é implementada pela ferramenta **Red Lizard Software Goanna**

4.2.2.3.1 Tipo de acesso

Em relação ao tipo de acesso, a ferramenta que implementa essa técnica reportou apenas ocorrências de *buffer overflow* em acessos de leitura de *buffer*. Entretanto, os valores das métricas dessa ferramenta, mesmo para esse tipo de acesso, foram consideravelmente baixas.

Dos 1470 casos de teste relacionados a *buffer overflows* ocorridos em acesso de escrita, a ferramenta que implementa a checagem de modelos não foi capaz de reportar nenhuma ocorrência da vulnerabilidade.

4.2.2.3.2 Local de acesso

No que se refere ao local de acesso onde ocorre a vulnerabilidade, a ferramenta reportou apenas ocorrências da vulnerabilidade no limite final do *buffer*, assemelhando-se, nesse contexto, à Análise de Fluxo de Dados e diferindo-se do uso conjunto de Lógica de Separação e *Shape Analysis*.

4.2.2.3.3 Tipo de alocação

4.2.2.4 Análise Estática Baseada em Padrões

A análise estática baseada em padrões é implementada pela ferramenta **Parasoft C++test**.

De modo geral, a **Parasoft C++test** foi a ferramenta com melhor desempenho entre as ferramentas analisadas. Informações adicionais sobre a ferramenta não puderam ser feitos por ser um software proprietário. O uso dessa técnica de análise estática é, a partir das informações levantadas sobre cada ferramenta, o fator que mais difere a **Parasoft C++test** das outras.

4.2.2.4.1 Tipo de acesso

Em relação ao tipo de acesso, a ferramenta teve uma maior eficiência em identificar *buffer overflows* ocorridos em acesso de leitura (como já foi evidenciado na Seção 4.2.1.3). Comparando-se esse esse resultado com os obtidos a partir das outras técnicas de análise, percebe-se que a diferença entre as métricas obtidas para cada tipo de acesso são maiores

para essa ferramenta. No contexto dessa técnica de análise, a diferença das taxas de *Recall* e *Discrim Rate* entre cada tipo de acesso são de cerca de 10%.

A partir desse resultado, pode-se inferir dois possíveis cenários: (i) A Análise Estática Baseada em Padrões é bem mais eficiente para detecção de *buffer overflow* quando este ocorre em acesso de leitura; (ii) A Análise Estática Baseada em Padrões é mais eficiente na detecção de *buffer overflows* ocorridos em acesso de leitura e essa eficiência é aprimorada pela Análise de Fluxo de Dados implementada pela ferramenta.

4.2.2.4.2 Local de acesso

No que se refere ao local do *buffer* onde ocorre a vulnerabilidade, a diferença entre as métricas obtidas para cada local de acesso é bem menor que a apresentada no cenário anterior, sendo cerca de 5% para *Recall* e *Discrim Rate* e 6% para *F-Score*. Como essa diferença favorece as métricas obtidas no subgrupo relacionado a *buffer overflows* ocorridos na parte final do *buffer* e este comportamento também ocorre nas ferramentas que implementam Análise de Fluxo de dados (que a **Parasoft** também implementa), não é possível afirmar, dentro do contexto de análise desse trabalho, que a Análise Estática Baseada em Padrões é mais eficiente na detecção de *buffer overflows* em acessos no final do *buffer* do que no início.

Tem-se, então, dois cenários prováveis: (i) A Análise Estática Baseada em Padrões não possui diferença considerável de eficiência em relação ao local de acesso onde ocorre o *buffer overflow*; (ii) A Análise Estática Baseada em Padrões, em conjunto com a Análise de Fluxo de de Dados, possui maior eficiência na detecção de *buffer overflows* quando o estes ocorrem na parte final do *buffer*.

4.2.2.4.3 Tipo de alocação

Em relação ao tipo de alocação de memória a **Parasoft** mostrou uma maior eficiência na detecção de *buffer overflows* em *buffers* alocados em *Heap*. Este comportamento reflete o cenário anterior: a **Parasoft**, que implementa Análise Estática Baseada em Padrões e Análise de Fluxo de Dados, apresentou comportamento semelhante às outras ferramentas analisadas que implementam Análise de Fluxo de Dados.

Em contrapartida, a diferença entre das métricas obtidas entre cada tipo de alocação é bem menor que a evidenciada no cenário anterior, sendo de menos de 3% para as métricas de *Recall* e *Discrim Rate*, e de pouco mais de 3% para a taxa de *F-Score*.

Dos cenários prováveis para este caso, tem-se o fato de que a Análise Estática Baseada em Padrões não tem influência relevante na detecção de *buffer overflows* ocorridos em um tipo de alocação, em detrimento de outro.

4.2.3 Análise da Combinação de Técnicas e de Ferramentas

A partir dos resultados apresentados na Análise das Métricas Obtidas por Ferramenta (Seção 4.2.1), e na Análise das Métricas Obtidas por Técnica de Análise (Seção 4.2.2), tem-se dados que podem ser utilizados como insumos numa possível integração de ferramentas. Esta seção tem por objetivo exemplificar possíveis caminhos, utilizando estes dados, para combinar uma ou mais ferramentas que implementem diferentes técnicas de análise de vulnerabilidade em software, visando uma maior eficiência na detecção de *buffer overflows*.

Como exemplo, foi feita uma análise conjunta dos resultados obtidos pelas ferramentas **Monoidics Infer** e **Red Lizard** nos casos de acesso de escrita e leitura de *buffer*, exibidos nas tabelas 16 e 17, respectivamente.

Dos 2352 casos de teste relacionados à ocorrência de *buffer overflow* em leitura de *buffer*, a tabela 16 mostra que a ferramenta **Monoidics Infer** foi capaz de detectar, sozinha, 84 Verdadeiros Positivos. Já a ferramenta **Red Lizard** foi capaz de detectar 24 Verdadeiros Positivos. As ferramentas não reportaram nenhuma ocorrência de Verdadeiro Positivo em comum.

Ferramenta/Valor	FN	FP	TN	TP
Monoidics	2268	0	2352	84
Monoidics com Redlizard	2244	0	2148	0
Somente Monoidics	24	0	204	84
Somente Redlizard	84	204	0	24
Redlizard	2328	204	2148	24

Tabela 16 – Buffer Overflows relacionados a Leitura de Buffer - 2352 Casos de teste

Analisando o valor obtido por cada ferramenta sobre o total de casos de teste existentes, tem-se que a ferramenta **Monoidics Infer** reportou 3,57% das ocorrências de Verdadeiro Positivo presentes. Já a ferramenta **Red Lizard** reportou, sozinha, 1,02% dessas ocorrências, apresentando uma diferença de 2,55% em relação à primeira ferramenta.

Ferramenta/Valor	FN	FP	TN	TP
monoidics	1386	0	1470	84
monoidics-redlizard	1386	0	1470	0
only_monoidics	0	0	0	84
only_redlizard	84	0	0	0
redlizard	1470	0	1470	0

Tabela 17 – Buffer Overflows relacionados a Escrita de Buffer - 1470 Casos de teste

Já a tabela 17, mostra que apenas a **Monoidics Infer** reportou verdadeiros positivos. Percentualmente, essa ferramenta reportou 5,7% das vulnerabilidades existentes (Verdadeiro Positivo) na suíte de testes.

A ferramenta **Red Lizard** mostrou, nos resultados obtidos no contexto desse trabalho, maior eficiência na detecção de *buffer overflows* em accesso de leitura de *buffer*, enquanto a ferramenta **Monoidics Infer** mostrou mais eficiência na detecção em acesso de escrita. Esses resultados estão dispostos na seção 4.2.1.

Na análise conjunta das duas ferramentas, a discrepância entre os resultados obtidos por cada uma reforça os fatos aqui evidenciados.

Parte III

Conclusão

5 Conclusão

Tendo considerado os resultados obtidos nesse trabalho e agrupados sob diversas perspectivas (por subgrupo de CWE, por ferramenta e por técnica de análise, (Seção 4.2), retorno às questões da Seção 1.5 que orientaram as pesquisas e análises descritas nesse documento para associá-las a esses resultados.

Em relação à primeira questão (**Quais os aspectos que caracterizam e identificam vulnerabilidades relacionadas a buffer overflow?**), o estudo das técnicas de análise estática está diretamente relacionado com o meio em que elas são aplicadas. Assim, para uma análise de cada técnica sobre a vulnerabilidade de *buffer overflow*, foi necessário um levantamento tórico acerca das características dessa vulnerabilidade. Este referencial foi levantado e aprofundado no Capítulo 2, (mais especificamente na seção 2.3.1).

A partir deste referencial e das características apresentadas pela vulnerabilidade, foi possível classificar subgrupos de *buffer overflow*, associando atributos relacionados à taxonomia da vulnerabilidade, com subgrupos de CWEs (seção 2.3.1.1).

Dessa forma, a perspectiva na qual se baseou esse trabalho caracteriza, a partir dos referenciais apresentados, a vulnerabilidade de *buffer overflow* em uma série de atributos que influenciam no modo como a vulnerabilidade se manifesta e, consequentemente, o modo como ela pode ser identificada em um determinado código.

Em relação à segunda questão (**Quais os principais aspectos que caracterizam cada abordagem de análise estática apresentada?**), foi feito, inicialmente, um breve levantamento sobre o funcionamento de cada técnica de análise descrita nesse trabalho (seção 2.2.1.2). Entretanto, as principais características que esse trabalho buscou evidenciar foram identificadas a partir de uma análise empírica e descritas no capítulo de Resultados (4).

Os resultados obtidos foram classificados por características do *buffer overflow* (seção 4.1) e análisados sob duas perspectivas: por ferramentas de análise (seção 4.2.1) e por técnica de análise, na seção Análise das métricas obtidas por técnica de análise (4.2.2).

Assim, dentro do contexto de análise apresentado nesse trabalho, cada técnica mostrou diferentes níveis de eficiência para cada subgrupo de *buffer overflow*, como evidenciado nas seções supracitadas.

Em relação à terceira questão (**Quais aspectos são redundantes e quais se somam, numa integração entre as abordagens apresentadas?**), tem-se a inte-

gração de ferramentas que implementem determinadas técnicas de vulnerabilidade como algo extremamente complexo. Duas ferramentas que reportem uma mesma ocorrência de vulnerabilidade dão mais confiabilidade pare esse resultado. Mas para uma análise precisa (com o mínimo de falsos positivos), é preciso extremo cuidado com as divergências entre os resultados dessas ferramentas.

A partir das análises feitas ao longo desse trabalho, foi feito um exemplo comparativo de ferramentas que implementam técnicas de análise diferentes (seção 4.2.3). Os resultados apresentados vão de acordo com as análises empíricas feitas aqui, evicenciando que a integração dos resultados de duas ou mais ferramentas mostra-se mais eficiente ou não de acordo com as características da vulnerabilidade apresentada.

Em relação à última questão levantada (**Que características são evidenciadas pelas métricas calculadas para cada tipo de abordagem?**), foi possível identificar, a partir dos resultados obtidos e agrupados por técnica de análise de vulnerabilidades, características comuns a cada técnica. Essas características, evidenciadas em Análise das métricas obtidas por técnica de análise (seção 4.2.2), foram em acordo aos resultados das ferramentas que implementam essas técnicas. Este fator, ainda que feito a partir de uma análise empírica, reforça a premissa de que cada técnica de análise de vulnerabilidade apresenta características, vantajosas ou não, diretamente ligadas às características da vulnerabilidade de *buffer overflow*.

Não foi intuito desse trabalho justificar os fatores que determinam as características de cada técnica de análise estática. As análises feitas possibilitaram somente identificar o modo como cada técnica se comporta de acordo com o contexto de análise.

Devido às limitações de dados para análise, as técnicas de Lógica de Separação e *Shape Analysis* não puderam ser analisadas separadamente. Logo, as características relacionadas a essas duas técnicas são apresentadas em conjunto. Não é possível inferir, no contexto desse trabalho, o comportamento de cada técnica de modo individual.

6 Trabalhos Futuros

As análises feitas nesse trabalho, e a contextualização teórica que embasou os resultados aqui apresentados, possibilitam o desenvolvimento de uma análise mais efetiva das vantagens e desvantagens de cada técnica de análise estática de vulnerabilidades.

Aliado a isso, o investimento de técnicas que mostrem-se efetivas em determinadas características de um *buffer overflow* permite que se obtenha uma maior eficiência na detecção de vulnerabilidades e, consequentemente, softwares que sejam mais seguros para o contexto em que se inserem.

Tanto para a construção de uma ferramenta mais robusta na detecção de uma ou mais vulnerabilidades em software, quanto para a integração de ferramentas para análise de um determinado código, desenvolvedores podem utilizar os resultados apresentados nesse trabalho e definir critérios para a seleção de ferramentas que se adequem aos seus interesses. Há ferramentas livres de análise estática de vulnerabilide que podem ser utilizadas terem seus resultados, sob um mesmo código, analisados.

A partir da sobreposição de ferramentas, é possível aumentar a confiabilidade da primeira ferramenta, nas áreas em que a segunda também reporta ocorrências da mesma vulnerabilidade. Além disso, nos contextos em que as duas ferramentas diferem, é possível avaliar qual ferramenta tem maior confiabilidade no reporte de uma ocorrência da vulnerabilidade em questão. Nesse caso, a eficiência de cada ferramenta na detecção de vulnerabilidades pode ser feita analisando-se, também, as características do código onde a mesma é reportada e avaliando a eficiência da ferramenta em questão para esse contexto.

O experimento realizado na Análise da Combinação de Técnicas e de Ferramentas (Seção 4.2.3) evidencia o potencial da análise conjunta de ferramentas para a melhorar a eficiência na detecção de vulnerabilidades, aumentando a quantidade de verdadeiros positivos reportados e diminuindo a quantidade falsos positivos e falsos negativos.

Supõe-se com base nesse experimento, o seguinte cenário: Tem-se que ferramenta **Red Lizard Software Goanna** implementa técnicas de maior eficiência na detecção de *buffer oveflow* que ocorra em acessos de leitura, enquanto a ferramenta **Monoidics Infer** tem um melhor desempenho na deteçcão dessa vulnerabilidade quando ela ocorre em acesso de escrita. Os casos em que os *buffer overflows* em acesso de leitura forem reportados pela ferramenta **Red Lizard** e não forem reportados pela ferramenta **Monoidics Infer**, devem ser avaliados considerando que a **Red Lizard** possui maior eficiência na detecção nesse contexto de ocorrência. Em um contexto em que ocorra acessos de escrita, o mesmo tipo de análise pode ser feito, considerando, nesse caso, a **Monoidics Infer** como ferramenta de maior eficiência. Além disso, caso as duas ferramentas reportem ocorrências

de *buffer overflow* em comum, esse resultado deve ser considerado ainda mais confiável, pois chegou-se a um mesmo resultado utilizando duas perspectivas diferentes. Um possível trabalho futuro, seria a verificação e validação da hipótese levantada nesse cenário, utilizando softwares reais como base de análise.

Ressalta-se que a análise das técnicas implementadas por cada ferramenta não deve ser o único fator de decisão sobre as ocorrências reportadas em um cenário como o apresentado, mas como um conhecimento adicional a ser utilizado nessa decisão. Diversos fatores podem influenciar o reporte correto e o tipo de análise apresentado nesse trabalho é indicativo e não determinante.

Referências

AGGARWAL, A.; JALOTE, P. Integrating static and dynamic analysis for detecting vulnerabilities. In: *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*. [S.l.: s.n.], 2006. v. 1, p. 343–350. ISSN 0730-3157. Citado na página 35.

ALMORSY, M.; GRUNDY, J.; IBRAHIM, A. S. Supporting automated vulnerability analysis using formalized vulnerability signatures. In: *ASE*. [S.l.: s.n.], 2012. p. 100–109. Citado na página 40.

CALCAGNO, C.; DISTEFANO, D. *Infer: An Automatic Program Verifier for Memory Safety of C Programs*. [S.l.], 2011. Último Acesso em: 2015-01-15. Disponível em: <http://www.eecs.qmul.ac.uk/~ddino/papers/nasa-infer.pdf>. Citado na página 59.

CAS. *CAS Static Analysis Tool Study - Methodology*. 9800 Savage Road, Fort George G. Meade, MD 20755-6738, 2011. Citado 5 vezes nas páginas 51, 52, 53, 55 e 66.

CHESS, B.; WEST, J. *Secure Programming with Static Analysis*. First. [S.l.]: Addison-Wesley Professional, 2007. ISBN 9780321424778. Citado na página 38.

CHRISTEY, S. *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. [S.l.], 2011. Disponível em: <http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf>. Citado na página 44.

COUSOT, P.; COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977. p. 238–252. Citado na página 42.

EADDY, M. et al. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In: *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2008. (ICPC '08), p. 53–62. ISBN 978-0-7695-3176-2. Disponível em: <http://dx.doi.org/10.1109/ICPC.2008.39>. Citado na página 43.

FEATURES - Red Lizard Software. 2009. <http://redlizards.com/features/>. Último Acesso em: 2014-10-15. Citado na página 60.

GROSSO, C. D. et al. Detecting buffer overflow via automatic test input data generation. *Comput. Oper. Res.*, Elsevier Science Ltd., Oxford, UK, UK, v. 35, n. 10, p. 3125–3143, out. 2008. ISSN 0305-0548. Disponível em: <http://dx.doi.org/10.1016/j.cor.2007.01.013>. Citado na página 31.

HOARE, C. A. R. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, v. 12, n. 10, p. 576–580, 1969. Citado na página 42.

JONES, R. L.; RASTOGI, A. Secure coding: Building security into the software development life cycle. In: . [S.l.: s.n.], 2004. p. 29–39. Citado 3 vezes nas páginas 35, 36 e 44.

KRATKIEWICZ, K.; LIPPMANN, R. A taxonomy of buffer overflows for evaluating static and dynamic software testing tools. In: *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*. [S.l.: s.n.], 2005. v. 1, n. 1, p. 44–51. Citado na página 44.

LDRA, T. *Static Analysis with LDRA Testbed*. [S.l.], 2011. Último Acesso em: 2015-01-15. Disponível em: <http://www.ldra.com/attachments/article/17/static-analysis-with-ldra.pdf>. Citado na página 58.

MARJAMAKI, D. *Cppcheck Design*. [S.l.], 2010. Último Acesso em: 2015-01-15. Disponível em: <http://ufpr.dl.sourceforge.net/project/cppcheck/Articles/cppcheck-design-2010.pdf>. Citado na página 58.

MUSKE, T. et al. Efficient elimination of false positives using bounded model checking. Pune, India, 2013., 2013. Citado na página 31.

O'HEARN, P. W. *A Primer on Separation Logic (and Automatic Program Verification and Analysis)*. [S.l.], 2012. Último Acesso em: 2015-01-15. Disponível em: <http://www0.cs.ucl.ac.uk/staff/p.ohearn/papers/Marktoberdorf11LectureNotes.pdf>. Citado 2 vezes nas páginas 42 e 59.

PARASOFT. *C++ Test DataSheet*. [S.l.], 2013. Último Acesso em: 2015-01-15. Disponível em: <http://www.parasoft.com/wp-content/uploads/pdf/C++TestDataSheet.pdf>. Citado na página 59.

PAUL, M. The need for secure software. In: . Palm Harbor, Florida: [s.n.], 2008. Citado 2 vezes nas páginas 27 e 35.

PRECHELT, L. *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program*. D-76128 Karlsruhe, Germany, 2000. Citado na página 61.

RöDIGER, W.-S. *Merging Static Analysis and Model Checking for Improved Security Vulnerability Detection*. Tese (Masters), 2011. Disponível em: <http://www.xn--wolfrdiger-icb.de/publication/roediger2011security.pdf>. Citado na página 30.

SHRESTHA, J. *Static Program Analysis*. 83 p. Dissertação (Mestrado) — Uppsala University, Information Systems, 2013. Citado 3 vezes nas páginas 27, 28 e 30.

TEIXEIRA, E. P. L. *Ferramenta de Análise de Código para Detecção de Vulnerabilidades*. Dissertação (Mestrado) — Engenharia Informática, Departamento de Informática, Universidade de Lisboa, 2007. Citado na página 38.

THE MITRE CORPORATION. *CWE Version 2.6*. Gaithersburg, MD, United States, 2014. Citado 2 vezes nas páginas 43 e 49.

XIE, Y.; AIKEN, A. Scalable error detection using boolean satisfiability. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 40, n. 1, p. 351–363, jan. 2005. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1047659.1040334>. Citado na página 32.

Apêndices

APÊNDICE A – SRAT - Sate Report Analysis Tool

O *Sate Report Analysis Tool* foi desenvolvido com o intuito extrair resultados de relatórios do SATE (*Static Analysis Tool Exposition*). Sua principal funcionalidade consiste em percorrer os resultados reportados para cada caso de teste e contabilizar o resultado obtido por cada ferramenta. Cada ferramenta pode reportar 4 tipos de resultado: TP (Verdadeiro Positivo), FP (Falso Positivo), FN (Falso Negativo) ou TN (Verdadeiro Negativo).

A figura 14 ilustra a estrutura de classes atual *parser*. O *parser* é composto por três módulos e um programa principal, todos caracterizados nas subseções a seguir.

A.0.0.0.1 Módulo *Extractor*

Módulo responsável por percorrer o relatório de dados no formato XML, extrair e contabilizar os resultados de cada ferramenta. Para percorrer a estrutura do arquivo XML, este módulo faz uso das bibliotecas perl XML::Parser e XML::SimpleObject. Os dados obtidos são agrupados por ferramenta e retornados para o módulo *Calculator*. Além da contagem dos resultados que cada ferramenta reporta, esse módulo também calculao número de *Discriminations* para cada subconjunto dado.

Caso a extração desejada seja a comparação entre duas ferramentas, este módulo separa separa os dados entre os resultados exclusivos de cada ferramenta e os resultados em comum entre elas.

A.0.0.0.2 Módulo *Calculator*

Este módulo utiliza os dados obtidos pelo módulo *Extractor* e calcula métricas a partir desses dados. Até o momento, são calculadas: a *False Positive Rate, Precision, Recall, F-Score* e *Discrim Rate*. As métricas calculadas são retornadas no formato de tabela (na linguagem **Perl**, uma tabela pode ser representada como uma *array* de *arrays*).

A.0.0.0.3 Módulo *Output*

Este módulo tem como funcionalidade a formatação de uma tabela para o formato CSV[1] ou LaTeX. Ele é independente dos módulos *Extractor* e *Calculator*, sendo responsável apenas por facilitar a interpretação dos dados e métricas calculados.

[1] http://creativyst.com/Doc/Articles/CSV/CSV01.htm

A.0.0.0.4 Módulo *Main*

O módulo *Main* é a interface de entrada do *parser*, recebendo as ferramentas e subgrupos de CWEs que serão utilizados e repassando-as ao *Calculator*. Após receber os dados calculados e métricas obtidos, o módulo *Main* repassa esses dados para o módulo *Output* e recebe esses dados no formato escolhido.

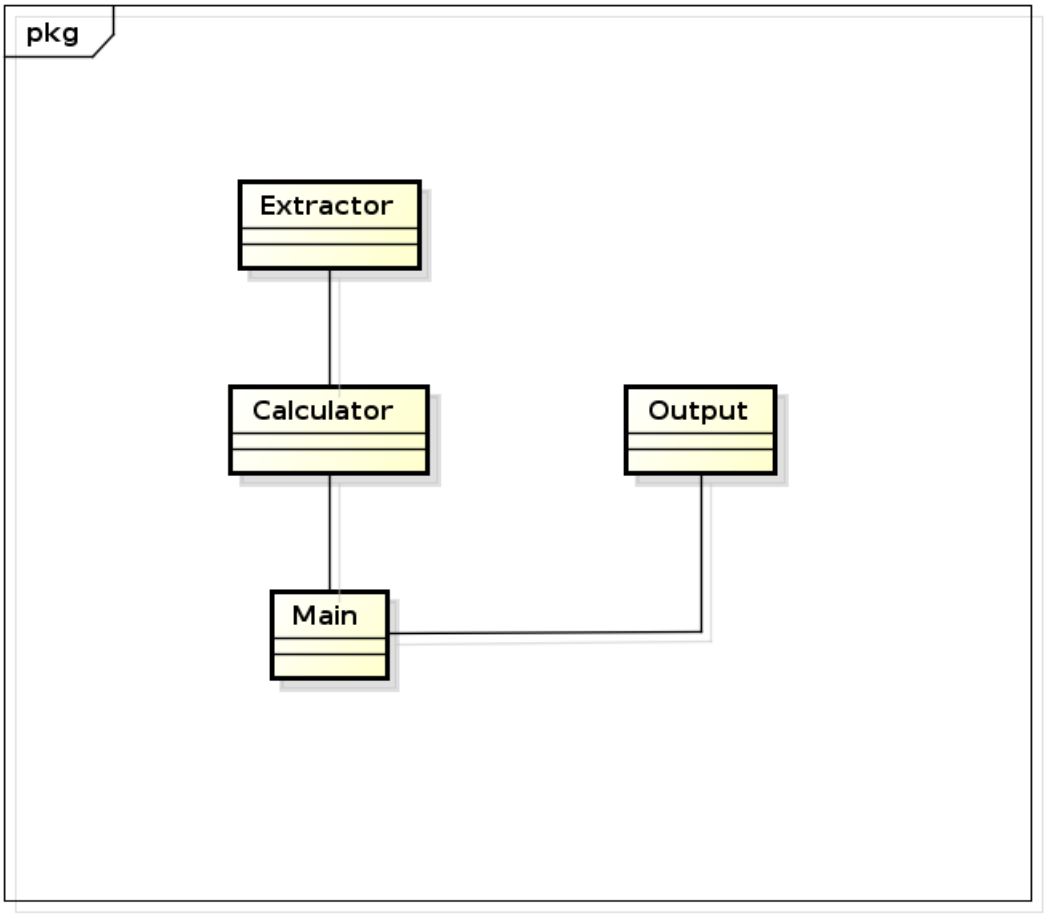

Figura 14 – Classes principais do *parser* de análise desenvolvido

Anexos

ANEXO A – CWEs relacionadas a *Buffer Overflow*

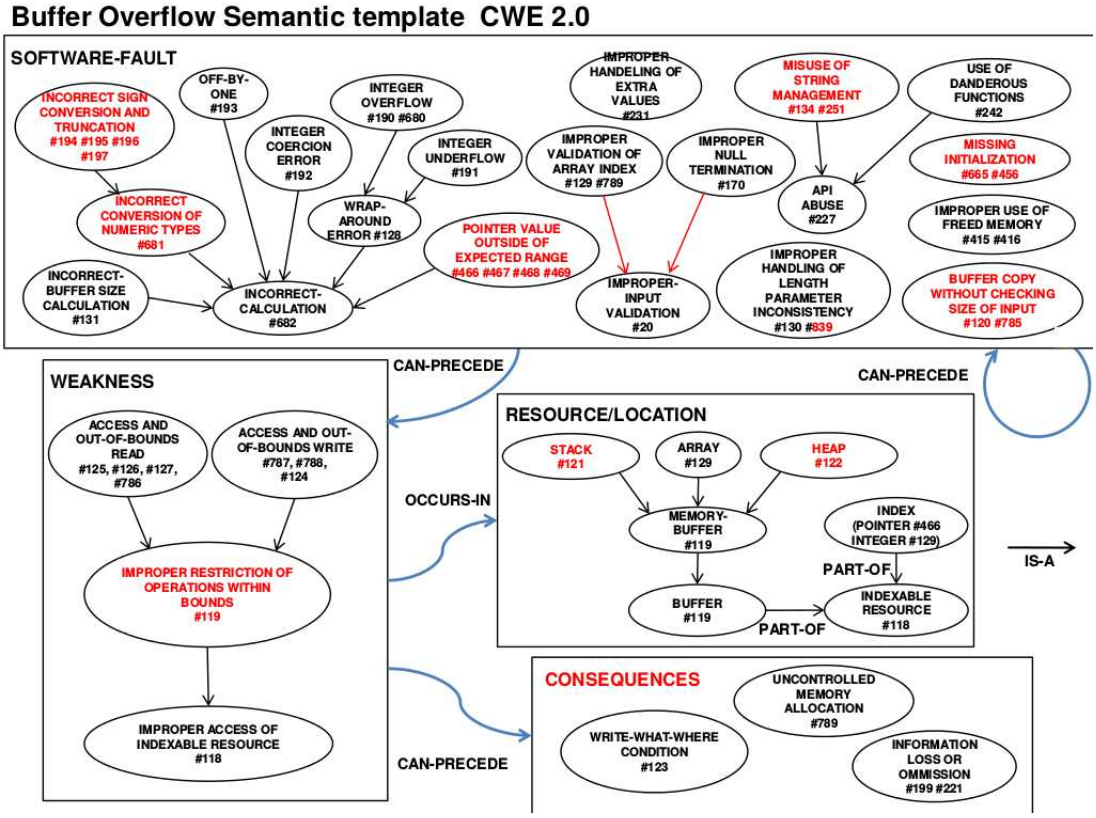

Figura 15 – *Buffer Overflow Semantic Template*

ANEXO B – Estrutura de *report* utilizada para análise

Listing B.1 – Código que pode resultar ou não em um *Buffer Overflow*

```
<?xml version="1.0" encoding="UTF-8"?>
<synthetic_results>
    <testcase syntcid="6384" cweid="121">
        <complexity type="Baseline"></complexity>
        <sources>
            <file>
            synthetic-c/testcases/CWE121_Stack_Based_Buffer_Overflow/
            CWE121_Stack_Based_Buffer_\
            Overflow__char_type_overrun_memcpy_01.c
            </file>
        </sources>
        <results>
            <participant name="cppcheck">
                <good status="true-negative"/>
                <bad status="false-negative"/>
            </participant>
            <participant name="ldra">
                <good status="true-negative"/>
                <bad status="false-negative"/>
            </participant>
            <participant name="monoidics">
                <good status="true-negative"/>
                <bad status="false-negative"/>
            </participant>
            <participant name="parasoft">
                <good status="true-negative"/>
                <bad status="false-negative"/>
            </participant>
            <participant name="redlizard">
                <good status="true-negative"/>
                <bad status="false-negative"/>
            </participant>
```

```
33          </results>
34      </testcase>
35      <testcase syntcid="12562" cweid="122">
36          <complexity type="Control">Dead code after a return</complexity>
37          <sources>
38              <file>
39              synthetic-c/testcases/CWE122_Heap_Based_Buffer_Overflow/
40              CWE122_Heap_Based_Buffer_\
41              Overflow__wchar_t_type_overrun_memmove_19.c
42              </file>
43          </sources>
44          <results>
45              <participant name="cppcheck">
46                  <good status="true-negative"/>
47                  <bad status="false-negative"/>
48              </participant>
49              <participant name="ldra">
50                  <good status="true-negative"/>
51                  <bad status="false-negative"/>
52              </participant>
53              <participant name="monoidics">
54                  <good status="true-negative"/>
55                  <bad status="false-negative"/>
56              </participant>
57              <participant name="parasoft">
58                  <good status="true-negative"/>
59                  <bad status="false-negative"/>
60              </participant>
61              <participant name="redlizard">
62                  <good status="true-negative"/>
63                  <bad status="false-negative"/>
64              </participant>
65          </results>
66      </testcase>
67  </synthetic_results>
```